

UNIVERSIDADE FEDERAL DE GOIÁS  
INSTITUTO DE INFORMÁTICA

# **Programação Orientada a Objetos**

Ciências da Computação

Herança

# Programação Orientada a Objetos

## Sumário

- Introdução
- Superclasses e Subclasses
- Membros protected
- Relacionamento entre Objetos de Superclasse e Objetos de Subclasse
- Construtores e Finalizadores em Subclasses
- Composição vs. Herança
- Estudo de Caso: Point, Circle, Cylinder
- Introdução ao Polimorfismo

# Herança

## Introdução

- Programação Orientada a Objetos
  - Acesso a membros **protected**
  - Relacionamentos
    - "é um" - herança
      - Objeto de subclasse "é um" objeto de superclasse
    - "tem um" - composição
      - Objeto "tem um" objeto de outra classe como um membro
  - Bibliotecas de classe
    - Novas classes podem herdar delas
    - Software pode ser construído de componentes padronizados e reutilizáveis
    - Cria softwares mais poderosos

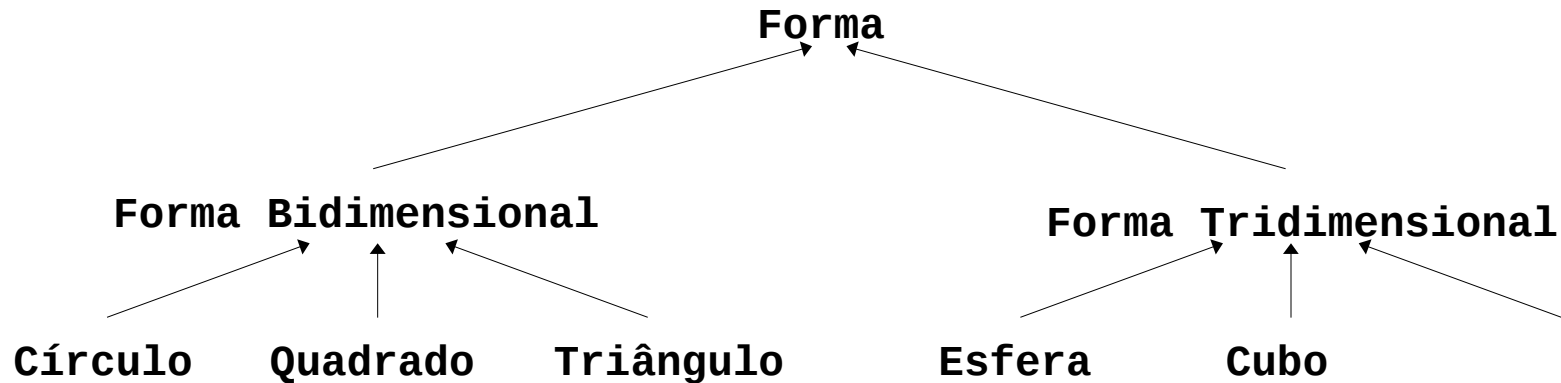
# Herança

## Superclasses e Subclasses

- Exemplo de herança
  - Um retângulo "é um" quadrilátero
    - Retângulo é um tipo específico de quadrilátero
    - Quadrilátero é a superclasse, retângulo é a subclasse
    - É incorreto dizer que quadrilátero "é um" retângulo
  - Subclasse tem mais características que superclasse
    - Subclasse é mais específica que superclasse
    - Toda subclasse "é um" objeto de sua superclasse, mas não o contrário
  - Estruturas hierárquicas em forma de árvore
    - Hierarquia para a classe **Forma** (próximo slide)

# Herança

## Superclasses e Subclasses



- Usando herança
  - Utiliza a palavra-chave **extends**  
**class FormaBidimensional extends Forma{ ... }**
  - membros **private** da superclasse não acessíveis diretamente às subclasses

# Herança

## Membros **protected**

- Em uma superclasse
  - membros **public**
    - Acessíveis por qualquer programa que tenha a referência de uma superclasse or subclasse
  - membros **private**
    - Acessíveis somente em métodos da superclasse
  - membros **protected**
    - Proteção intermediária entre **private** e **public**
    - Acessíveis somente por métodos de superclasse, de subclasse, ou classes no mesmo pacote
- Métodos de subclasse
  - Pode referenciar membros **public** ou **protected** pelo nome
  - Métodos sobrepostos acessíveis com **super.nomeMétodo**

# Herança

## Objetos de Superclasse e de Subclasse

- Objeto de subclasse
  - Pode ser tratado como objeto de superclasse
    - O contrário não é verdadeiro
  - Supondo que muitas classes herdam de uma superclasse
    - Array de referências à superclasse pode ser definido
    - Todos os objetos são tratados como objetos da superclasse
  - Coerção (cast) explícita
    - Converte referências à superclasse para uma referência à subclasse (downcasting)
    - Feito quando a referência à superclasse realmente referencia um objeto da subclasse
  - operador **instanceof**
    - **if (p instanceof Circulo)**
    - Retorna **true** se o objeto para o qual **p** aponta "é um" **Circulo**

# Herança

## Objetos de Superclasse e de Subclasse

- Sobrepondo métodos
  - Subclasse pode redefinir métodos da superclasse
    - Quando o método é mencionado na subclasse, a versão da subclasse é utilizada
    - O acesso ao método original da superclasse é feito com **super.nomeMetodo**
  - Para invocar o construtor da superclasse explicitamente
    - **super(); //argumentos podem ser passados**
    - Se chamado explicitamente, deve ser a primeira declaração



# Herança

## Objetos de Superclasse e de Subclasse

- Uma Applet usa estas técnicas
  - Conceito de herança formalizado
  - Java usa implicitamente a classe **Object** como superclasse para todas as classes
  - Os métodos **init** e **paint** das applets são herdados da classe **JApplet**

# Herança

## Objetos de Superclasse e de Subclasse

- Exemplo

```
4 public class Point {
```

- Toda classe herda implicitamente da classe **Object**

```
8     public Point()
```

- Todo construtor deve chamar o construtor da superclasse
  - Por padrão (default), é chamado implicitamente
  - Se explícito, deve ser a primeira declaração

```
41 public class Circle extends Point { // herda de Point
```

- Herda de **Point**

```
54     super( a, b ); // chama o construtor da superclass
```

- Chama explicitamente o construtor da superclasse (**Point**)
  - Deve ser a primeira declaração no construtor **Circle**
  - Para chamar o construtor padrão, use **super()**

# Herança

## Objetos de Superclasse e de Subclasse

- **Point** e **Circle** sobrepõem **toString**
  - Para chamar **toString**, da classe **Point**, na classe **Circle**, use **super.toString()**

```
83    Point pointRef, p;  
84    Circle circleRef, c;  
  
88    c = new Circle( 2.7, 120, 89 );  
  
95    pointRef = c;    // atribui Circle a pointRef
```

- **pointRef** aponta para um objeto **Circle**
  - É permitido pois **Circle** "é um" **Point**

```
98    pointRef.toString();
```

- **pointRef** sabe que está apontando para um objeto **Circle**
  - Chama o método **toString** apropriado
  - Exemplo de polimorfismo

# Herança

## Objetos de Superclasse e de Subclasse

```
102         circleRef = (Circle) pointRef;
```

- **pointRef** está apontando para um **Circle**
  - Realiza um downcast para uma referência a **Circle** e atribui a **circleRef**

```
87         p = new Point( 30, 50 );  
113         if ( p instanceof Circle ) {
```

- Operador **instanceof**
  - Retorna **true** se o objeto para o qual **p** aponta "é um" **Circle**
  - Neste exemplo, retorna **false**

```

1 // Fig. 9.4: Point.java
2 // Definition of class Point
3
4 public class Point {
5     protected int x, y; // coordinates of the Point
6
7     // No-argument constructor
8     public Point()
9     {
10         // implicit call to superclass constructor occurs here
11         setPoint( 0, 0 );
12     }
13
14     // Constructor
15     public Point( int a, int b )
16     {
17         // implicit call to superclass constructor occurs here
18         setPoint( a, b );
19     }
20
21     // Set x and y coordinates of Point
22     public void setPoint( int a, int b )
23     {
24         x = a;
25         y = b;
26     }
27
28     // get x coordinate
29     public int getX() { return x; }

```

Point implicitamente herda da classe  
**Object**

Note o construtor padrão (sem  
argumentos). Chamada implícita ao  
construtor da superclasse.

```

30
31 // get y coordinate
32 public int getY() { return y; }
33
34 // convert the point into a String representation
35 public String toString()
36     { return "[" + x + ", " + y + "]" ; }
37 }
38 // Fig. 9.4: Circle.java
39 // Definition of class Circle
40
41 public class Circle extends Point { // inherits from Point
42     protected double radius;
43
44     // No-argument constructor
45     public Circle()
46     {
47         // implicit call to superclass constructor occurs here
48         setRadius( 0 );
49     }
50
51     // Constructor
52     public Circle( double r, int a, int b )
53     {
54         super( a, b ); // call to superclass
55         setRadius( r );
56     }
57
58     // Set radius of Circle
59     public void setRadius( double r )
60     { radius = ( r >= 0.0 ? r : 0.0 ); }

```

Método **toString** da classe **Point** sobrepõe o **toString** original na classe **Object**

Construtor **Point** chamado implicitamente

Construtor **Point** chamado explicitamente. Deve ser a primeira declaração no construtor da subclasse

```
61
62 // Get radius of Circle
63 public double getRadius() { return radius; }
64
65 // Calculate area of Circle
66 public double area() { return Math.PI * radius * radius; }
67
68 // convert the Circle to a String
69 public String toString()
70 {
71     return "Center = " + "[" + x + ", " + y + "]" +
72           "; Radius = " + radius;
73 }
74 }
```

```

75 // Fig. 9.4: Test.java
76 // Demonstrating the "is a" relationship
77 import java.text.DecimalFormat;
78 import javax.swing.JOptionPane;
79
80 public class InheritanceTest {
81     public static void main( String args[] )
82     {
83         Point pointRef, p;
84         Circle circleRef, c;
85         String output;
86
87         p = new Point( 30, 50 );
88         c = new Circle( 2.7, 120, 89 );
89
90         output = "Point p: "
91                 + "\nCircle c:
92
93         // use the "is a" relationship
94         // with a Point reference
95         pointRef = c; // assign ci
96
97         output += "\n\nCircle c (via pointRef): " +
98                 pointRef.toString();

```

Permitido pois um objeto da subclasse "é um" objeto da superclasse

O computador sabe que **pointRef** está realmente apontando para um objeto **Circle**, daí ele chama o método **toString** da classe **Circle**. Este é um exemplo de polimorfismo.



```

99
100 // Use downcasting (casting a superclass reference to a
101 // subclass data type) to assign pointRef to circleRef
102 circleRef = (Circle) pointRef;
103
104 output += "\n\nCircle c (via circleRef): " +
105         circleRef.toString();
106
107 DecimalFormat precision2 = new DecimalFormat(
108 output += "\nArea of c (via circleRef): " +
109         precision2.format( circleRef.area() ),
110
111 // Attempt to refer to Point object
112 // with Circle reference
113 if ( p instanceof Circle ) {
114     circleRef = (Circle) p; // line 40 in Test.java
115     output += "\n\ncast successful";
116 }
117 else
118     output += "\n\np does not refer to a Circle";
119
120 JOptionPane.showMessageDialog( null, output,
121     "Demonstrating the \"is a\" relationship",
122     JOptionPane.INFORMATION_MESSAGE );
123
124 System.exit( 0 );
125 }
126 }

```

Realiza o downcast de **pointRef** (o qual está realmente apontando para um **Circle**) para um **Circle**, e o atribui para **circleRef**

Testa se **p** (classe **Point**) "é um" **Circle**. Ele não é.



## Demonstrating the "is a" relationship



Point p: [30, 50]

Circle c: Center = [120, 89]; Radius = 2.7

Circle c (via pointRef): Center = [120, 89]; Radius = 2.7

Circle c (via circleRef): Center = [120, 89]; Radius = 2.7

Area of c (via circleRef): 22.90

p does not refer to a Circle

OK

# Herança

## Construtores e Finalizadores

- Objetos de subclasse
  - O construtor de superclasse deve ser chamado
    - Inicializa variáveis da superclasse
  - Construtor padrão é chamado implicitamente
  - Chamada explícita (usando **super**) deve ser a primeira declaração
- Se o método **finalize** é definido
  - O método **finalize** da subclasse deve chamar o método **finalize** da superclasse, como última ação
  - Deve ser **protected**

```
23    protected void finalize()
```

```

1 // Fig. 9.5: Point.java
2 // Definition of class Point
3 public class Point extends Object {
4     protected int x, y; // coordinates of the Point
5
6     // no-argument constructor
7     public Point()
8     {
9         x = 0;
10        y = 0;
11        System.out.println( "Point constructor: " + this );
12    }
13
14    // constructor
15    public Point( int a, int b )
16    {
17        x = a;
18        y = b;
19        System.out.println( "Point constructor: " + this );
20    }
21
22    // finalizer
23    protected void finalize()
24    {
25        System.out.println( "Point finalizer: " + this );
26    }
27
28    // convert the point into a String representation
29    public String toString()
30    { return "[" + x + ", " + y + "]; }
31 }

```

Somente subclasses podem acessar métodos **protected**.

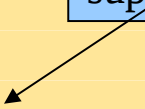


```

32 // Fig. 9.5: Circle.java
33 // Definition of class Circle
34 public class Circle extends Point { // inherits from Point
35     protected double radius;
36
37     // no-argument constructor
38     public Circle()
39     {
40         // implicit call to superclass constructor here
41         radius = 0;
42         System.out.println( "Circle constructor: " + this );
43     }
44
45     // Constructor
46     public Circle( double r, int a, int b )
47     {
48         super( a, b ); // call the superclass constructor
49         radius = r;
50         System.out.println( "Circle constructor: " + this );
51     }
52
53     // finalizer
54     protected void finalize()
55     {
56         System.out.println( "Circle finalizer: " + this );
57         super.finalize(); // call superclass finalize method
58     }
59

```

Método finalize de **Circle**  
chama o finalizador da  
superclasse (em **Point**).



```
60 // convert the Circle to a String
61 public String toString()
62 {
63     return "Center = " + super.toString() +
64         "; Radius = " + radius;
65 }
66 }
```

```
67 // Fig. 9.5: Test.java
```

```
68 // Demonstrate when superclass and subclass
```

```
69 // constructors and finalizers are called.
```

```
70 public class Test {
```

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

```
72     {
```

```
73         Circle circle1, circle2;
```

```
74         circle1 = new Circle( 4.5,
```

```
75         circle2 = new Circle( 10, 5, 5 );
```

```
76
```

```
77
```

```
78         circle1 = null; // mark for garbage collection
```

```
79         circle2 = null; // mark for garbage collection
```

```
80
```

```
81         System.gc(); // call the garbage collector
```

```
82     }
```

```
83 }
```

Marca objetos para a coleta de lixo, estabelecendo-os em **null**.

Chama o coletor de lixo (método **static gc** da classe **System**).

```
Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5
Point constructor: Center = [5, 5]; Radius = 0.0
Circle constructor: Center = [5, 5]; Radius = 10.0
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [5, 5]; Radius = 10.0
Point finalizer: Center = [5, 5]; Radius = 10.0
```

# Herança

## Conversão de Objetos

- Referências
  - Referências a objetos de subclasse podem ser implicitamente convertidas para referências de superclasse
    - Subclasse contém membros correspondentes àqueles da subclasse
  - Referenciando um objeto de subclasse com uma referência de superclasse
    - Permitido: um objeto de subclasse "é um" objeto de superclasse
    - Somente membros da superclasse podem ser referenciados
  - Referenciando um objeto de superclasse com uma referência de subclasse
    - Erro
    - Primeiramente deve ser feito uma coerção (cast) para uma referência de superclasse



# Composição vs. Herança

- Relacionamento "é um"
  - Herança
- Relacionamento "tem um"
  - Composição, tendo outros objetos como membros
- Exemplo

**Empregado "é um" DataAniversario;      //ERRADO!**

**Empregado "tem um" DataAniversario;      //Composição**

# Herança

## Estudo de Caso

- Exemplo de herança
  - Classe **Point**
    - variáveis **x, y** **protected**
    - Métodos: **setPoint, getX, getY, toString**
  - Classe **Circle** (**extends Point**)
    - variável **radius** **protected**
    - Métodos: **setRadius, getRadius, area, toString** sobreposto
  - Classe **Cylinder** (**extends Circle**)
    - variável **height** **protected**
    - Métodos: **setHeight, getHeight, area** (área da superfície), **volume, toString** sobreposto

```
1 // Fig. 9.6: Point.java
2 // Definition of class Point
3 package com.deitel.jhttp3.ch09;
4
5 public class Point {
6     protected int x, y; // coordinates of the Point
7
8     // no-argument constructor
9     public Point() { setPoint( 0, 0 ); }
10
11    // constructor
12    public Point( int a, int b ) { setPoint( a, b ); }
13
14    // Set x and y coordinates of Point
15    public void setPoint( int a, int b )
16    {
17        x = a;
18        y = b;
19    }
20
21    // get x coordinate
22    public int getX() { return x; }
23
24    // get y coordinate
25    public int getY() { return y; }
26
27    // convert the point into a String representation
28    public String toString()
29    { return "[" + x + ", " + y + "]"; }
30 }
```

```
1 // Fig. 9.7: Circle.java
2 // Definition of class Circle
3 package com.deitel.ihtp3.ch09:
4
5 public class Circle extends Point { // inherits from Point
6     protected double radius;
7
8     // no-argument constructor
9     public Circle()
10    {
11        // implicit call to superclass constructor
12        setRadius( 0 );
13    }
14
15    // Constructor
16    public Circle( double r, int a, int b )
17    {
18        super( a, b ); // call the superclass constructor
19        setRadius( r );
20    }
21
22    // Set radius of Circle
23    public void setRadius( double r )
24    { radius = ( r >= 0.0 ? r : 0.0 ); }
25
26    // Get radius of Circle
27    public double getRadius() { return radius; }
28
29    // Calculate area of Circle
30    public double area()
31    { return Math.PI * radius * radius; }
```

```
32
33 // convert the Circle to a String
34 public String toString()
35 {
36     return "Center = " + super.toString() +
37           "; Radius = " + radius;
38 }
39 }
```

```
1 // Fig. 9.8: Cylinder.java
2 // Definition of class Cylinder
3 package com.deitel.jhtp3.ch09;
4
5 public class Cylinder extends Circle {
6     protected double height; // height of Cylinder
7
8     // No-argument constructor
9     public Cylinder()
10    {
11        // implicit call to superclass constructor here
12        setHeight( 0 );
13    }
14
15    // constructor
16    public Cylinder( double h, double r, int a, int b )
17    {
18        super( r, a, b );
19        setHeight( h );
20    }
21
22    // Set height of Cylinder
23    public void setHeight( double h )
24        { height = ( h >= 0 ? h : 0 ); }
25
26    // Get height of Cylinder
27    public double getHeight() { return height; }
28
```

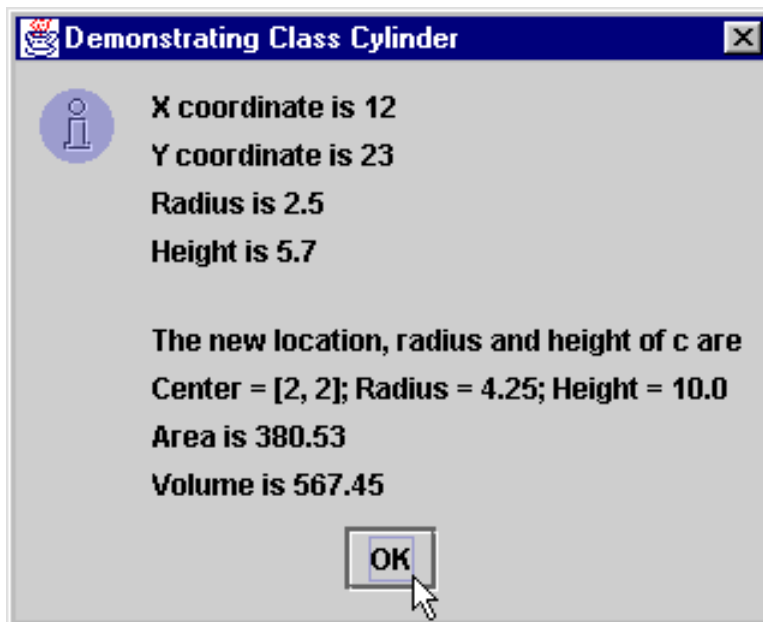
```

29 // Calculate area of Cvylinder (i.e., surface area)
30 public double area()
31 {
32     return 2 * super.area() +
33           2 * Math.PI * radius * height;
34 }
35
36 // Calculate volume of Cylinder
37 public double volume() { return super.area() * height; }
38
39 // Convert the Cvylinder to a String
40 public String toString()
41 {
42     return super.toString() + "; Height = " + height;
43 }
44 }
45 // Fig. 9.8: Test.java
46 // Application to test class Cylinder
47 import javax.swing.JOptionPane;
48 import java.text.DecimalFormat;
49 import com.deitel.ihtp3.ch09.Cvylinder;
50
51 public class Test {
52     public static void main( String args[] )
53     {
54         Cylinder c = new Cylinder( 5.7, 2.5, 12, 23 );
55         DecimalFormat precision2 = new DecimalFormat( "0.00" );
56         String output;
57

```

```
58     output = "X coordinate is " + c.getX() +
59             "\nY coordinate is " + c.getY() +
60             "\nRadius is " + c.getRadius() +
61             "\nHeight is " + c.getHeight();
62
63     c.setHeight( 10 );
64     c.setRadius( 4.25 );
65     c.setPoint( 2, 2 );
66
67     output +=
68         "\n\nThe new location, radius " +
69         "and height of c are\n" + c +
70         "\nArea is " + precision2.format( c.area() ) +
71         "\nVolume is " + precision2.format( c.volume() );
72
73     JOptionPane.showMessageDialog( null, output,
74         "Demonstrating Class Cylinder",
75         JOptionPane.INFORMATION_MESSAGE );
76     System.exit( 0 );
77 }
78 }
```





# Polimorfismo

## Introdução

- Polimorfismo
  - Programas extensíveis
  - Processamento de objetos de superclasse de forma genérica
  - Facilita a adição de classes à hierarquia
    - Pouca ou nenhuma modificação requerida
    - Apenas partes do programa que necessitam conhecimento direto da nova classe devem ser alteradas

# Polimorfismo

## Vinculação Dinâmica de Métodos

- Vinculação dinâmica de métodos
  - Em tempo de execução, a chamada ao método é direcionada à versão apropriada
  - Método da classe apropriada é chamado
- Exemplo
  - **Triângulo**, **Círculo**, e **Quadrado** são subclasses de **Forma**
    - Cada uma delas tem um método **desenha**
  - A chamada a **desenha** usa referências de superclasse
    - Em tempo de execução, o programa determina para qual classe a referência está realmente apontando
    - Chama o método **desenha** apropriado

# Polimorfismo

## Métodos e Classes **final**

- Declarando variáveis **final**
  - Elas não podem ser modificadas após a declaração
  - Devem ser inicializadas na declaração
- Declarando métodos **final**
  - Não pode ser sobreposto em uma subclasse
  - métodos **static** e **private** são implicitamente **final**
- Declarando classes **final**
  - Não pode ser uma superclasse (novas classes não podem ser derivadas a partir dela)
  - Todos os métodos na classe são implicitamente **final**

# Polimorfismo

## Superclasses Abstratas e Concretas

- Classes Abstratas (superclasses abstratas)
  - Único propósito é ser uma superclasse
    - Outras classes herdam dela
  - Não é possível instanciar objetos de uma classe abstrata
    - Variáveis de instância e construtores continuam a existir
  - Classe declarada com a palavra-chave **abstract**
- Classe concreta
  - É possível instanciar objetos
  - Específica
- Hierarquias de classe
  - Classes gerais são usualmente **abstract**
    - **FormaBidimensional** - muito genérica para ser concreta

# Polimorfismo

## Exemplos

- Classe **Quadrilátero**
  - **Retângulo** "é um" **Quadrilátero**
  - método **getPerimetro** pode ser invocado por qualquer subclasse
    - **Quadrado, Paralelogramo, Trapezóide**
    - O mesmo método possui "muitas formas" - polimorfismo
  - Imagine um vetor (*array*) de referências à superclasse
    - O *array* poderia conter todos os objetos
  - Invocação a **getPerimetro** é feita usando as referências
    - O método apropriado para cada classe é invocado
- Adicionando uma nova subclasse
  - Necessário apenas definir **getPerimetro** para aquela nova classe
  - A referência à nova classe pode ser feita com a referência a superclasse

# Polimorfismo

## Exemplos

- Com o polimorfismo
  - Novas classes podem ser adicionadas facilmente
  - A invocação a um método pode resultar em diferentes ações, dependendo do objeto que recebe a invocação
- Referências
  - Referências podem ser criadas para classes abstratas
    - Não é permitida a instanciação de objetos de classes abstratas
- métodos abstratos
  - palavra-chave **abstract**
    - Uma classe com método **abstract** deve ser **abstract**
  - métodos **abstract** devem ser sobrepostos na subclasse
    - Se isso não ocorre, a subclasse deve ser **abstract**

# Polimorfismo

## Exemplos

- Classes iteradoras
  - Varrem todos os objetos em um *container* (como um *array*, por exemplo)
  - Utilizado em programação polimórfica
    - Varre um *array* de referências à superclasse
    - Invoca o método **desenha** para cada referência



# Polimorfismo

## Estudo de Caso: Folha de Pagamentos

- Programa-exemplo
  - superclasse abstrata **Employee**
    - método abstrato **earnings**
      - Deve ser implementado em cada subclasse
  - Classes **Boss**, **CommissionWorker**, e **PieceWorker**, **HourlyWorker**
    - Sobrepoem os métodos **toString** e **earnings**
  - Classe **Test**
    - Inicializa objetos
    - Usa uma referência a **Employee** e invoca **toString** e **earnings**
    - Através do polimorfismo, os métodos apropriados de cada classes são invocados

```

1 // Fig. 9.9: Employee.java
2 // Abstract base class Employee
3
4 public abstract class Employee {
5     private String firstName;
6     private String lastName;
7
8     // Constructor
9     public Employee( String first, String last ) {
10
11         firstName = first;
12         lastName = last;
13     }
14
15     // Return the first name
16     public String getFirstName() { return firstName; }
17
18     // Return the last name
19     public String getLastName() { return lastName; }
20
21     public String toString()
22     { return firstName + ' ' + lastName; }
23
24     // Abstract method that must be implemented for each
25     // derived class of Employee from which objects
26     // are instantiated.
27     public abstract double earnings();
28 }

```

Classe **Employee** declarada como abstrata, portanto nenhum objeto **Employee** pode ser criado.

**Employee** pode ter ainda um construtor, usado por suas classes derivadas.

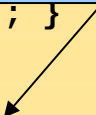
Métodos abstratos devem ser definidos em subclasses concretas.

```

29 // Fig. 9.9: Boss.java
30 // Boss class derived from Employee
31
32 public final class Boss extends Employee {
33     private double weeklySalary;
34
35     // Constructor for class Boss
36     public Boss( String first, String last, double s)
37     {
38         super( first, last ); // call superclass constructor
39         setWeeklySalary( s );
40     }
41
42     // Set the Boss's salary
43     public void setWeeklySalary( double s )
44     { weeklySalary = ( s > 0 ? s : 0 ); }
45
46     // Get the Boss's pay
47     public double earnings() { return weeklySalary; }
48
49     // Print the Boss's name
50     public String toString()
51     {
52         return "Boss: " + super.toString();
53     }
54 }

```

A implementação de **earnings** é requerida pois ele foi declarado como abstrato e **Boss** é uma classe concreta.



```
55 // Fig. 9.9: CommissionWorker.java
56 // CommissionWorker class derived from Employee
57
58 public final class CommissionWorker extends Employee {
59     private double salary;        // base salary per week
60     private double commission;    // amount per item sold
61     private int quantity;         // total items sold for week
62
63     // Constructor for class CommissionWorker
64     public CommissionWorker( String first, String last,
65                             double s, double c, int q)
66     {
67         super( first, last ); // call superclass constructor
68         setSalary( s );
69         setCommission( c );
70         setQuantity( q );
71     }
72
73     // Set CommissionWorker's weekly base salary
74     public void setSalary( double s )
75     { salary = ( s > 0 ? s : 0 ); }
76
77     // Set CommissionWorker's commission
78     public void setCommission( double c )
79     { commission = ( c > 0 ? c : 0 ); }
80
81     // Set CommissionWorker's quantity sold
82     public void setQuantity( int q )
83     { quantity = ( q > 0 ? q : 0 ); }
84
```

```

85 // Determine CommissionWorker's earnings
86 public double earnings()
87     { return salary + commission * quantity; }
88
89 // Print the CommissionWorker's name
90 public String toString()
91 {
92     return "Commission worker: " + super.toString();
93 }
94 }
95 // Fig. 9.9: PieceWorker.java
96 // PieceWorker class derived from Employee
97
98 public final class PieceWorker extends Employee {
99     private double wagePerPiece; // wage per piece output
100     private int quantity;        // output for week
101
102     // Constructor for class PieceWorker
103     public PieceWorker( String first, String last,
104                        double w, int q )
105     {
106         super( first, last ); // call superclass constructor
107         setWage( w );
108         setQuantity( q );
109     }
110
111     // Set the wage
112     public void setWage( double w )
113     { wagePerPiece = ( w > 0 ? w : 0 ); }
114

```

```
115// Set the number of items output
116    public void setQuantity( int q )
117        { quantity = ( q > 0 ? q : 0 ); }
118
119    // Determine the PieceWorker's earnings
120    public double earnings()
121        { return quantity * wagePerPiece; }
122
123    public String toString()
124    {
125        return "Piece worker: " + super.toString();
126    }
127}
128// Fig. 9.9: HourlyWorker.java
129// Definition of class HourlyWorker
130
131public final class HourlyWorker extends Employee {
132    private double wage;    // wage per hour
133    private double hours;   // hours worked for week
134
135    // Constructor for class HourlyWorker
136    public HourlyWorker( String first, String last,
137                        double w, double h )
138    {
139        super( first, last );    // call superclass constructor
140        setWage( w );
141        setHours( h );
142    }
143
```

```

144 // Set the wage
145 public void setWage( double w )
146     { wage = ( w > 0 ? w : 0 ); }
147
148 // Set the hours worked
149 public void setHours( double h )
150     { hours = ( h >= 0 && h < 168 ? h : 0 ); }
151
152 // Get the HourlyWorker's pay
153 public double earnings() { return wage * hours; }
154
155 public String toString()
156 {
157     return "Hourly worker: " + super.toString();
158 }
159}

```

160// Fig. 9.9: Test.java

161// Driver for Employee hierarchy

162import javax.swing.JOptionPane;

163import java.text.DecimalFormat;

164

165public class Test {

166 public static void main( String args[] )

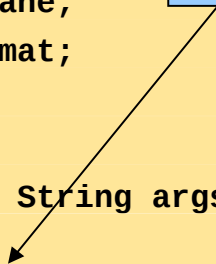
167 {

168 Employee ref; // superclass reference

169 String output = "";

170

Referências a classes abstratas  
são permitidas.



```

171 Boss b = new Boss( "John", "Smith", 800.00 );
172 CommissionWorker c =
173     new CommissionWorker( "Sue", "Jones",
174                           400.0, 3.0, 150);
175 PieceWorker p =
176     new PieceWorker(
177     HourlyWorker h =
178     new HourlyWorker( "Karen", 11.00, 10.75, 40 );
179
180 DecimalFormat precision2 = new DecimalFormat( "0.00" );
181
182 ref = b; // Employee reference to a Boss
183 output += ref.toString() + " earned $" +
184           precision2.format( ref.earnings() ) + "\n" +
185           b.toString() + " earned $" +
186           precision2.format( b.earnings() ) + "\n";
187
188 ref = c; // Employee reference to a CommissionWorker
189 output += ref.toString() + " earned $" +
190           precision2.format( ref.earnings() ) + "\n" +
191           c.toString() + " earned $" +
192           precision2.format( c.earnings() ) + "\n";
193
194 ref = p; // Employee reference to a PieceWorker
195 output += ref.toString() + " earned $" +
196           precision2.format( ref.earnings() ) + "\n" +
197           p.toString() + " earned $" +
198           precision2.format( p.earnings() ) + "\n";
199

```

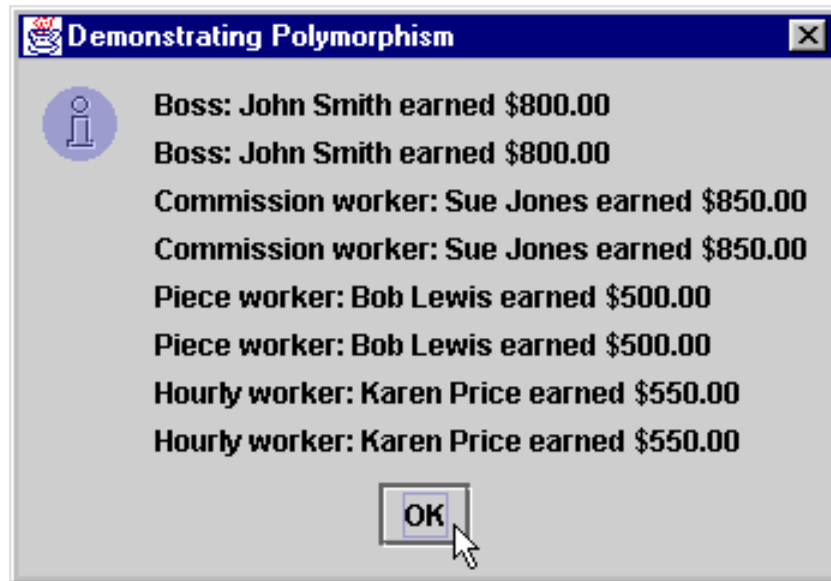
A referência a Employee **ref** aponta para o objeto **Boss**.  
Através do polimorfismo, as versões apropriadas de **toString** e **earnings**, da classe **Boss**, são invocadas.



```

200     ref = h; // Employee reference to an HourlyWorker
201     output += ref.toString() + " earned $" +
202               precision2.format( ref.earnings() ) + "\n" +
203               h.toString() + " earned $" +
204               precision2.format( h.earnings() ) + "\n";
205
206     JOptionPane.showMessageDialog( null, output,
207                                   "Demonstrating Polymorphism",
208                                   JOptionPane.INFORMATION_MESSAGE );
209     System.exit( 0 );
210 }
211 }

```



# Novas Classes e Alocação Dinâmica

- Alocação Dinâmica
  - Acomoda novas classes
  - Tipo do objeto não é requerido em tempo de compilação
  - Em tempo de execução, a invocação ao método é combinada com o objeto

# Polimorfismo

## Estudo de Caso: Herdando Interface e Implementação

- Exemplo de Polimorfismo
  - superclasse abstrata **Shape**
    - Subclasses **Point**, **Circle**, **Cylinder**
    - método abstrato
      - **getName**
    - métodos não-abstratos
      - **area (return 0.0)**
      - **volume (return 0.0)**
  - Classe **Shape** usada para definir um conjunto de métodos comuns
    - Interface é composta dos três métodos comuns
    - Implementação da **area** e **volume** usada nos primeiros níveis da hierarquia

# Polimorfismo

## Estudo de Caso: Herdando Interface e Implementação

- Criação de um *array* de referências a **Shape**
  - Aponta para várias referências a **Point**
  - Invoca métodos utilizando referências a **Shape**

```
159         arrayOfShapes[ i ].getName() + ": " +  
160         arrayOfShapes[ i ].toString() +
```

```

1 // Fig. 9.10: Shape.java
2 // Definition of abstract base class Shape
3
4 public abstract class Shape extends Object {
5     public double area() { return 0.0; }
6     public double volume() { return 0.0; }
7     public abstract String getName();
8 }
9 // Fig. 9.10: Point.java
10 // Definition of class Point
11
12 public class Point extends Shape {
13     protected int x, v; //
14
15     // no-argument constructor
16     public Point() { setPoint( 0, 0 ); }
17
18     // constructor
19     public Point( int a, int b ) { setPoint( a, b ); }
20
21     // Set x and v coordinates of Point
22     public void setPoint( int a, int b )
23     {
24         x = a;
25         v = b;
26     }
27
28     // get x coordinate
29     public int getX() { return x; }
30

```


Métodos **area** e **volume** são definidos.  
Eles serão sobrepostos nas subclasses quando necessário.

```

31 // get y coordinate
32 public int getY() { return y; }
33
34 // convert the point into a String representation
35 public String toString()
36     { return "[" + x + ", " + y + "]" ; }
37
38 // return the class name
39 public String getName() { return "Point"; }
40 }
41 // Fig. 9.10: Circle.java
42 // Definition of class Circle
43
44 public class Circle extends Point { // inherits from Point
45     protected double radius;
46
47     // no-argument constructor
48     public Circle()
49     {
50         // implicit call to superclass constructor here
51         setRadius( 0 );
52     }
53
54     // Constructor
55     public Circle( double r, int a, int b )
56     {
57         super( a, b ); // call the superclass constructor
58         setRadius( r );
59     }
60

```

Método abstrato **getName** deve ser sobreposto em uma subclasse concreta.



```

61 // Set radius of Circle
62 public void setRadius( double r )
63     { radius = ( r >= 0 ? r : 0 ); }
64
65 // Get radius of Circle
66 public double getRadius() { return radius; }
67
68 // Calculate area of Circle
69 public double area() { return Math.PI * radius * radius; }
70
71 // convert the Circle to a String
72 public String toString()
73     { return "Center = " + super.
74         "; Radius = " + radi
75
76 // return the class name
77 public String getName() { return "Circle"; }
78 }
79 // Fig. 9.10: Cvllinder.iava
80 // Definition of class Cvllinder
81
82 public class Cvllinder extends Circle {
83     protected double height; // height of Cvllinder
84
85     // no-argument constructor
86     public Cvllinder()
87     {
88         // implicit call to superclass constructor here
89         setHeight( 0 );
90     }

```

**Circle** sobrepõe o método **area**, herdado de **Shape**. **Point** não sobrepõe **area**, e tem a implementação padrão (returns **0**).

```
91
92 // constructor
93 public Cvylinder( double h, double r, int a, int b )
94 {
95     super( r, a, b ); // call superclass constructor
96     setHeight( h );
97 }
98
99 // Set height of Cvylinder
100 public void setHeight( double h )
101     { height = ( h >= 0 ? h : 0 ); }
102
103 // Get height of Cvylinder
104 public double getHeight() { return height; }
105
106 // Calculate area of Cvylinder (i.e., surface area)
107 public double area()
108 {
109     return 2 * super.area() +
110         2 * Math.PI * radius * height;
111 }
112
113 // Calculate volume of Cvylinder
114 public double volume() { return super.area() * height; }
115
116 // Convert a Cvylinder to a String
117 public String toString()
118     { return super.toString() + "; Height = " + height; }
119
120 // Return the class name
121 public String getName() { return "Cvylinder"; }
122 }
```



```
123// Fig. 9.10: Test.java
124// Driver for point, circle, cylinder hierarchy
125import javax.swing.JOptionPane;
126import java.text.DecimalFormat;
127
128public class Test {
129    public static void main( String args[] )
130    {
131        Point point = new Point( 7, 11 );
132        Circle circle = new Circle( 3.5, 22,
133        Cylinder cylinder = new Cylinder( 10
134
135        Shape arrayOfShapes[];
136
137        arrayOfShapes = new Shape[ 3 ];
138
139        // aim arrayOfShapes[0] at subclass Point object
140        arrayOfShapes[ 0 ] = point;
141
142        // aim arrayOfShapes[1] at subclass Circle object
143        arrayOfShapes[ 1 ] = circle;
144
145        // aim arrayOfShapes[2] at subclass Cylinder object
146        arrayOfShapes[ 2 ] = cylinder;
147
148        String output =
149            point.getName() + ": " + point.toString() + "\n" +
150            circle.getName() + ": " + circle.toString() + "\n" +
151            cylinder.getName() + ": " + cylinder.toString();
```

Cria um *array* de referências a **Shape**.

As referências a **Shape** aponta para objetos de subclasse.

```

152
153     DecimalFormat precision2 = new DecimalFormat( "0.00" );
154
155     // Loop through arrayOfShapes and print the name,
156     // area, and volume of each object.
157     for ( int i = 0; i < arrayOfShapes.length; i++ ) {
158         output += "\n\n" +
159             arrayOfShapes[ i ].getName() + ": " +
160             arrayOfShapes[ i ].toString() +
161             "\nArea = " +
162             precision2.format( arrayOfShapes[ i ].getArea() ) +
163             "\nVolume = " +
164             precision2.format( arrayOfShapes[ i ].getVolume() ) +
165     }
166
167     JOptionPane.showMessageDialog( null, output,
168         "Demonstrating Polymorphism",
169         JOptionPane.INFORMATION_MESSAGE );
170
171     System.exit( 0 );
172 }
173 }

```

Invoca métodos usando referências a superclasse. O método apropriado é invocado graças ao polimorfismo.



## Demonstrating Polymorphism



**Point: [7, 11]**

**Circle: Center = [22, 8]; Radius = 3.5**

**Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0**

**Point: [7, 11]**

**Area = 0.00**

**Volume = 0.00**

**Circle: Center = [22, 8]; Radius = 3.5**

**Area = 38.48**

**Volume = 0.00**

**Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0**

**Area = 275.77**

**Volume = 342.12**



# Polimorfismo

## Estudo de Caso: Criando e Utilizando Interfaces

- Interface
  - Palavra-chave **interface**
  - Possui um conjunto de métodos **public abstract**
  - Pode conter dados **public final static**
- Usando interfaces
  - Classe específica que usa interfaces com a palavra-chave **implements**
  - Classe deve definir todos os métodos abstratos existentes na interface
    - Deve usar o mesmo número de argumentos, mesmo tipo de retorno
  - Usar interfaces é como assinar um contrato
    - “Eu definirei todos os métodos especificados na interface”
  - O mesmo que o relacionamento “é um” da herança

# Polimorfismo

## Estudo de Caso: Criando e Utilizando Interfaces

- Usando interfaces
  - Interfaces são usadas no lugar de classes abstratas
    - Usadas quando não há nenhuma implementação padrão
  - Tipicamente tipos de dados públicos
    - Interface definida em seu próprio arquivo **.java**
    - Nome da interface é o mesmo nome do arquivo

# Polimorfismo

## Estudo de Caso: Criando e Utilizando Interfaces

- Outro uso para interfaces
    - Definir um conjunto de constantes usadas por muitas classes
- ```
public interface Constantes {  
    public static final int UM = 1;  
    public static final int DOIS = 2;  
    public static final int TRES = 3;  
}
```
- Reexaminando hierarquia anterior
    - Substituindo a classe abstrata **Shape** pela interface **Shape**

```
1 // Fig. 9.11: Shape.java
2 // Definition of interface Shape
3
4 public interface Shape {
5     public abstract double area();
6     public abstract double volume();
7     public abstract String getName();
8 }
```


Interface **Shape** está no seu próprio arquivo **Shape.java**

```
9 // Fig. 9.11: Point.java
10 // Definition of class Point
11
12 public class Point extends Object implements Shape {
13     protected int x, y; // coordinates of the Point
14
15     // no-argument constructor
16     public Point() { setPoint( 0, 0 ); }
17
18     // constructor
19     public Point( int a, int b ) { setPoint( a, b ); }
20
21     // Set x and y coordinates of Point
22     public void setPoint( int a, int b )
23     {
24         x = a;
25         y = b;
26     }
27 }
```

**Point** implementa a interface **Shape**, portanto é necessária a definição dos métodos abstratos de **Shape**.

```
28 // get x coordinate
29 public int getX() { return x; }
30
31 // get y coordinate
32 public int getY() { return y; }
33
34 // convert the point into a String representation
35 public String toString()
36     { return "[" + x + ", " + y +
37
38 // return the area
39 public double area() { return 0.0; }
40
41 // return the volume
42 public double volume() { return 0.0; }
43
44 // return the class name
45 public String getName() { return "Point"; }
46 }
```

**Point** precisa definir os métodos abstratos da interface que ele implementa.





```

123// Fig. 9.11: Test.java
124// Driver for point, circle, cylinder hierarchy
125import javax.swing.JOptionPane;
126import java.text.DecimalFormat;
127
128public class Test {
129    public static void main( String args[] )
130    {
131        Point point = new Point( 7, 11 );
132        Circle circle = new Circle( 3.5, 22, 8 );
133        Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
134
135        Shape arrayOfShapes[];
136
137        arrayOfShapes = new Shape[ 3 ];
138
139        // aim arrayOfShapes[0] at subclass Point object
140        arrayOfShapes[ 0 ] = point;
141
142        // aim arrayOfShapes[1] at subclass Circle object
143        arrayOfShapes[ 1 ] = circle;
144
145        // aim arrayOfShapes[2] at subclass Cylinder object
146        arrayOfShapes[ 2 ] = cylinder;
147
148        String output =
149            point.getName() + ": " + point.toString() + "\n" +
150            circle.getName() + ": " + circle.toString() + "\n" +
151            cylinder.getName() + ": " + cylinder.toString();

```

**Usa a mesma classe  
Test como antes**

```

152
153     DecimalFormat precision2 = new DecimalFormat( "0.00" );
154
155     // Loop through arrayOfShapes and print the name,
156     // area, and volume of each object.
157     for ( int i = 0; i < arrayOfShapes.length; i++ ) {
158         output += "\n\n" +
159             arrayOfShapes[ i ].getName() + ": " +
160             arrayOfShapes[ i ].toString() +
161             "\nArea = " +
162             precision2.format( arrayOfShapes[ i ].area() ) +
163             "\nVolume = " +
164             precision2.format( arrayOfShapes[ i ].volume() );
165     }
166
167     JOptionPane.showMessageDialog( null, output,
168         "Demonstrating Polymorphism",
169         JOptionPane.INFORMATION_MESSAGE );
170
171     System.exit( 0 );
172 }
173 }

```

**Usa a mesma classe  
Test como antes**



## Demonstrating Polymorphism



**Point: [7, 11]**

**Circle: Center = [22, 8]; Radius = 3.5**

**Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0**

**Point: [7, 11]**

**Area = 0.00**

**Volume = 0.00**

**Circle: Center = [22, 8]; Radius = 3.5**

**Area = 38.48**

**Volume = 0.00**

**Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0**

**Area = 275.77**

**Volume = 342.12**

OK

# Definições de Classe Interna

- Classes Internas (inner classes)
  - Até agora todas as classes foram definidas no escopo do arquivo (nenhuma dentro de outras classes)
  - Classes internas são definidas dentro de outras classes
    - Podem acessar todos os membros de classes externas
    - Nenhuma manipulação especial é necessária
  - Classes internas anônimas
    - Não possuem nome
  - Utilizadas frequentemente para manusear eventos

# Definições de Classe Interna

- Exemplo
  - Usa a classe **Time** e executa uma aplicação em sua própria janela
  - Usa tratamento de eventos para configurar a hora
    - Define um classe interna que implementa **ActionListener**
    - Usa um objeto desta classe como o tratador de eventos

```
1 // Fig. 9.12: Time.java
2 // Time class definition
3 import java.text.DecimalFormat; // used for number formatting
4
5 // This class maintains the time in 24-hour format
6 public class Time extends Object {
7     private int hour;        // 0 - 23
8     private int minute;     // 0 - 59
9     private int second;     // 0 - 59
10
11     // Time constructor initializes each instance variable
12     // to zero. Ensures that Time object starts in a
13     // consistent state.
14     public Time() { setTime( 0, 0, 0 ); }
15
16     // Set a new time value using universal time. Perform
17     // validity checks on the data. Set invalid values to zero.
18     public void setTime( int h, int m, int s )
19     {
20         setHour( h );        // set the hour
21         setMinute( m );     // set the minute
22         setSecond( s );     // set the second
23     }
24
25     // set the hour
26     public void setHour( int h )
27     { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }
28
29     // set the minute
30     public void setMinute( int m )
31     { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }
```

```
32
33 // set the second
34 public void setSecond( int s )
35     { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }
36
37 // get the hour
38 public int getHour() { return hour; }
39
40 // get the minute
41 public int getMinute() { return minute; }
42
43 // get the second
44 public int getSecond() { return second; }
45
46 // Convert to String in standard-time format
47 public String toString()
48 {
49     DecimalFormat twoDigits = new DecimalFormat( "00" );
50
51     return ( ( getHour() == 12 || getHour() == 0 ) ?
52         12 : getHour() % 12 ) + ":" +
53         twoDigits.format( getMinute() ) + ":" +
54         twoDigits.format( getSecond() ) +
55         ( getHour() < 12 ? " AM" : " PM" );
56 }
57 }
```

```

58 // Fig. 9.12: TimeTestWindow.java
59 // Demonstrating the Time class set and get methods
60 import java.awt.*;
61 import java.awt.event.*;
62 import javax.swing.*;
63
64 public class TimeTestWindow {
65     private Time t;
66     private JLabel hourLabel;
67     private JTextField hourField;
68     private JTextField secondField;
69     private JButton exitButton;
70
71     public TimeTestWindow()
72     {
73         super( "Inner Class Demonstration" );
74
75         t = new Time();
76
77         Container c = getContentPane();
78
79         // create an instance of the inner class
80         ActionEvent handler = new ActionEvent();
81
82         c.setLayout( new FlowLayout() );
83         hourLabel = new JLabel( "Set Hour" );
84         hourField = new JTextField( 10 );
85         hourField.addActionListener( handler );
86         c.add( hourLabel );
87         c.add( hourField );
88

```

Configura todos os componentes GUI no construtor. No **main**, cria um objeto **TimeTestWindow** que executa o construtor.

Invoca o construtor de **JFrame** para definir a barra de título da janela.

Objeto da classe interna **ActionEventHandler** passado como argumento para **addActionListener**



```
89     minuteLabel = new JLabel( "Set minute" );
90     minuteField = new JTextField( 10 );
91     minuteField.addActionListener( handler );
92     c.add( minuteLabel );
93     c.add( minuteField );
94
95     secondLabel = new JLabel( "Set Second" );
96     secondField = new JTextField( 10 );
97     secondField.addActionListener( handler );
98     c.add( secondLabel );
99     c.add( secondField );
100
101     display = new JTextField( 30 );
102     display.setEditable( false );
103     c.add( display );
104
105     exitButton = new JButton( "Exit" );
106     exitButton.addActionListener( handler );
107     c.add( exitButton );
108 }
109
110 public void displayTime()
111 {
112     display.setText( "The time is: " + t );
113 }
114
115 public static void main( String args[] )
116 {
117     TimeTestWindow window = new TimeTestWindow();
```

Cria objeto **TimeTestWindow**.

A blue rectangular callout box with a black border contains the text "Cria objeto TimeTestWindow.". A black arrow originates from the bottom-left corner of the box and points to the line of code "TimeTestWindow window = new TimeTestWindow();" in the code block.

```

118
119     window.setSize( 400, 140 );
120     window.show();
121 }
122
123 // INNER CLASS DEFINITION FOR EVENT HANDLING
124 private class ActionEventHandler implements ActionListener {
125     public void actionPerformed((ActionEvent e)
126     {
127         if ( e.getSource() == exitButton )
128             System.exit( 0 ); // terminate the application
129         else if ( e.getSource() == hourField ) {
130             t.setHour(
131                 Integer.parseInt( e.getActionCommand() ) );
132             hourField.setText( "" );
133         }
134         else if ( e.getSource() == minuteField ) {
135             t.setMinute(
136                 Integer.parseInt( e.getActionCommand() ) );
137             minuteField.setText( "" );
138         }
139         else if ( e.getSource() == secondField ) {
140             t.setSecond(
141                 Integer.parseInt( e.getActionCommand() ) );
142             secondField.setText( "" );
143         }
144
145         displayTime();
146     }
147 }
148 }

```

Use uma classe interna para tratar eventos. Classe implementa interface **ActionListener**, e deve definir **actionPerformed**.

**Inner Class Demonstration**

Set Hour  Set Sec

The time is: 1:00:00 PM

Exit

**Inner Class Demonstration**

Set Hour  Set minute  Set Sec

The time is: 1:00:00 PM

Exit

**Inner Class Demonstration**

Set Hour  Set minute  Set Second

The time is: 1:26:00 PM

Exit

**Inner Class Demonstration**

Set Hour  Set minute  Set Second

The time is: 1:00:00 PM

Exit

**Inner Class Demonstration**

Set Hour  Set minute  Set Second

The time is: 1:26:00 PM

Exit

**Inner Class Demonstration**

Set Hour  Set minute  Set Second

The time is: 1:26:07 PM

Exit

# Definições de Classe Interna

- Aplicações em Janelas
  - Executa uma aplicação em sua própria janela (com uma applet)

```
64 public class TimeTestWindow extends JFrame {
```

- Herda da classe **JFrame** (**javax.swing**) em vez de **JApplet**
  - Invoca construtor para definir título:  
**super( "TitleName" )**
- método **init** substituído pelo construtor
  - **init** somente é invocado para Applets
  - componentes GUI são definidos no construtor

```
117         TimeTestWindow window = new TimeTestWindow();
```

- Instanciação de objetos é feita no **main**

# Definições de Classe Interna

- Janela



# Definições de Classe Interna

```
123      // INNER CLASS DEFINITION FOR EVENT HANDLING
124      private class ActionEventHandler implements ActionListener {
```

- Tratamento de eventos
- Algumas classes devem implementar a interface **ActionListener**
  - Deve definir método **actionPerformed**
  - Classe que implementa **ActionListener** “é um” **ActionListener**

```
79      // create an instance of the inner class
80      ActionEventHandler handler = new ActionEventHandler();
85      hourField.addActionListener( handler );
```

- Método **addActionListener**
  - Usa objeto do tipo **ActionListener**

# Definições de Classe Interna

- Classes internas anônimas
  - Não têm nome
  - Objeto criado na definição da classe
- Tratamento de eventos com classes internas anônimas
  - Define a classe interna dentro da invocação ao método **addActionListener**
    - Cria uma instância da classe dentro da invocação ao método
    - **addActionListener** usa um objeto da classe **ChangeListener**

# Definições de Classe Interna

```
24    hourField.addActionListener(  
25        new ActionListener() { // anonymous inner class  
26            public void actionPerformed( ActionEvent e )  
27            {  
28                t.setHour(  
29                    Integer.parseInt( e.getActionCommand() ) );  
30                hourField.setText( "" );  
31                displayTime();  
32            }  
33        }  
34    );
```

- **new** cria um objeto
- **ActionListener()**
  - Inicia a definição da classe anônima e invoca construtor
  - Similar a

**public class myHandler implements ActionListener**

- Chaves ( **{}** ) começam e terminam a definição da classe



# Definições de Classe Interna

```
82    TimeTestWindow window = new TimeTestWindow();
83
84    window.addWindowListener(
85        new WindowAdapter() {
86            public void windowClosing( WindowEvent e )
87            {
88                System.exit( 0 );
89            }
90        }
91    );
```

- Enables o uso do tratador de eventos para janelas
  - **addWindowListener** registra um “escutador” para eventos associados a janelas
  - 7 métodos precisam ser definidos para a interface **WindowListener**
  - Classe Adapter – já implementa a interface
  - Classe Adapter “é um” **WindowListener**

# Definições de Classe Interna

```
82    TimeTestWindow window = new TimeTestWindow();
84    window.addWindowListener(
85        new WindowAdapter() {
86            public void windowClosing( WindowEvent e )
87            {
88                System.exit( 0 );
89            }
90        }
91    );
```

## – new **WindowAdapter()**

- Inicia uma classe interna anônima que herda de **WindowAdapter**
- Similar a:


```
public class meuTratadorDeEventos extends
    WindowAdapter {
```

```

1 // Fig. 9.13: TimeTestWindow.java
2 // Demonstrating the Time class set and get methods
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class TimeTestWindow extends JFrame {
8     private Time t;
9     private JLabel hourLabel, minuteLabel, secondLabel;
10    private JTextField hourField, minuteField,
11                secondField, display;
12
13    public TimeTestWindow()
14    {
15        super( "Inner Class Demonstration" );
16
17        t = new Time();
18
19        Container c = getContentPane();
20
21        c.setLayout( new FlowLayout() );
22        hourLabel = new JLabel( "Set Hour" );
23        hourField = new JTextField( 10 );
24        hourField.addActionListener(
25            new ActionListener() { // anonymous inner class
26                public void actionPerformed((ActionEvent e)
27                {
28                    t.setHour(
29                        Integer.parseInt( e.getActionCommand() ) );

```

Um objeto de uma classe interna anônima é usada como tratador de eventos.




```

30         hourField.setText( "" );
31         displayTime();
32     }
33 }
34 );
35 c.add( hourLabel );
36 c.add( hourField );
37
38 minuteLabel = new JLabel( "Set minute" );
39 minuteField = new JTextField( 10 );
40 minuteField.addActionListener(
41     new ActionListener() { // anonymous inner class
42         public void actionPerformed((ActionEvent e)
43         {
44             t.setMinute(
45                 Integer.parseInt( e.getActionCommand() ) );
46             minuteField.setText( "" );
47             displayTime();
48         }
49     }
50 );
51 c.add( minuteLabel );
52 c.add( minuteField );
53
54 secondLabel = new JLabel( "Set Second" );
55 secondField = new JTextField( 10 );
56 secondField.addActionListener(
57     new ActionListener() { // anonymous inner class
58         public void actionPerformed((ActionEvent e)

```

Cada objeto JTextField tem uma classe interna separada para tratar os eventos.




```
59         {
60             t.setSecond(
61                 Integer.parseInt( e.getActionCommand() ) );
62             secondField.setText( "" );
63             displayTime();
64         }
65     }
66 );
67 c.add( secondLabel );
68 c.add( secondField );
69
70 display = new JTextField( 30 );
71 display.setEditable( false );
72 c.add( display );
73 }
74
75 public void displayTime()
76 {
77     display.setText( "The time is: " + t );
78 }
79
80 public static void main( String args[] )
81 {
82     TimeTestWindow window = new TimeTestWindow();
83 }
```

```
84     window.addListener(  
85         new WindowAdapter() {  
86             public void windowClosing( WindowEvent e )  
87             {  
88                 System.exit( 0 );  
89             }  
90         }  
91     );  
92  
93     window.setSize( 400, 120 );  
94     window.show();  
95 }  
96 }
```


Usa uma classe interna  
anônima para tratar eventos  
janela.

### 3.2 addWindowListener

 **Inner Class Demonstration** [-] [ ] [X]

Set Hour  Set minute


Set Second

 **Inner Class Demonstration** [-] [ ] [X]

Set Hour  Set minute

Set Second


The time is: 7:00:00 AM

 **Inner Class Demonstration** [-] [ ] [X]

Set Hour  Set minute

Set Second

The time is: 7:00:00 AM

 **Inner Class Demonstration** [-] [ ] [X]

Set Hour  Set minute

Set Second

The time is: 7:13:00 AM



# Definições de Classe Interna

- Notas
  - Cada classe (inclusive classes internas) tem seu próprio arquivo **.class**
  - Classes internas com nome podem ser **public**, **protected**, **private**, ou acesso de pacote
  - Para acessar a referência **this** da classe externa
    - **NomeClasseExterna.this**
  - Classe interna pode ser estática (**static**)
    - Não requer que objeto da classe externa seja definido
    - Não tem acesso membros não-estáticos da classe externa



# Definições de Classe Interna

- Notas
  - Para criar um objeto de outra classe interna da classe
    - Criar um objeto da classe externa e atribuir a ele uma referência (**ref**)
    - Escrever a declaração na forma:

*NomeClasseExterna* . *NomeClasseInterna* **intRef** =  
**ref.new** *NomeClasseInterna* ( ) ;