

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Programação Orientada a Objetos

Ciências da Computação

Fundamentos de Orientação a Objetos

Sumário

8.1 Introdução

8.2 Implementando um Tipo Abstrato de Dados Time com uma Classe

8.3 Escopo de Classe

8.4 Controlando Acesso aos Membros

8.5 Criando Pacotes

8.6 Inicializando Objetos: Construtores

8.7 Usando Construtores Sobrecarregados

8.8 Usando Métodos *Set* e *Get*

8.9 Reutilização de Software

8.10 Variáveis de Instância Final

8.11 Composição: Objetos como Variáveis de Instância de Outras Classes

8.12 Acesso a Pacote

8.13 Usando a referência *this*

8.14 Finalizadores

8.15 Membros de Classe Estática

O que é a Orientação a Objetos ? ...

Classe → **tipo** ou **molde**

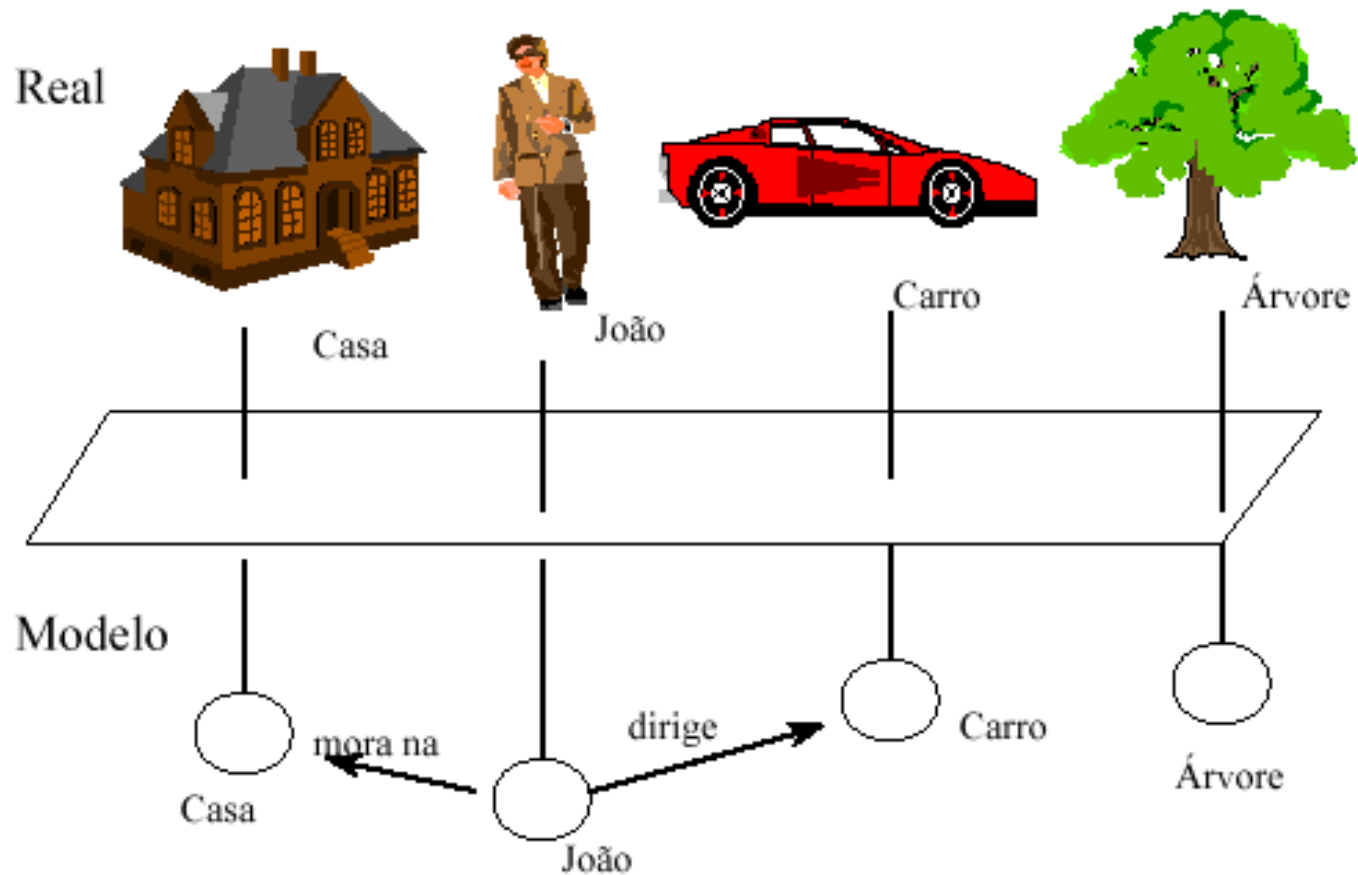


Objeto → **instância** ou **exemplar**

Mensagem → **comunicação** entre objetos

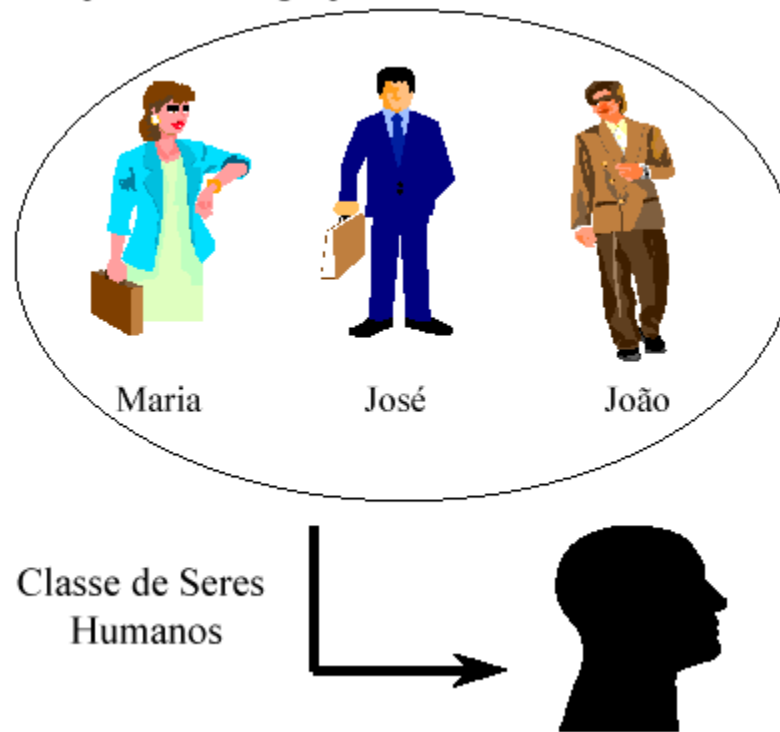
vocabulário básico

Tecnologia de Orientação a Objetos



Tecnologia de Orientação a Objetos

Classe: representa um “gabarito” para muitos objetos e descreve como estes objetos estão estruturados internamente.



Tecnologia de Orientação a Objetos

Objetos: “coisas” do mundo real



Uma **instância** é um **objeto** criado a partir de uma classe.
A **classe** descreve a estrutura da instância, enquanto que o estado da **instância** é definido pelas **operações** realizadas sobre ela.

Tecnologia de Orientação a Objetos

Propriedade e Atributos: objetos do mundo real possuem propriedades e valores para estas propriedades

	<u>Maria</u>	<u>José</u>	<u>João</u>
• idade	31	28	43
• endereço	Rua xx	Rua yy	Av zz
• sexo	Fem	Masc	Masc
• etc			



Maria



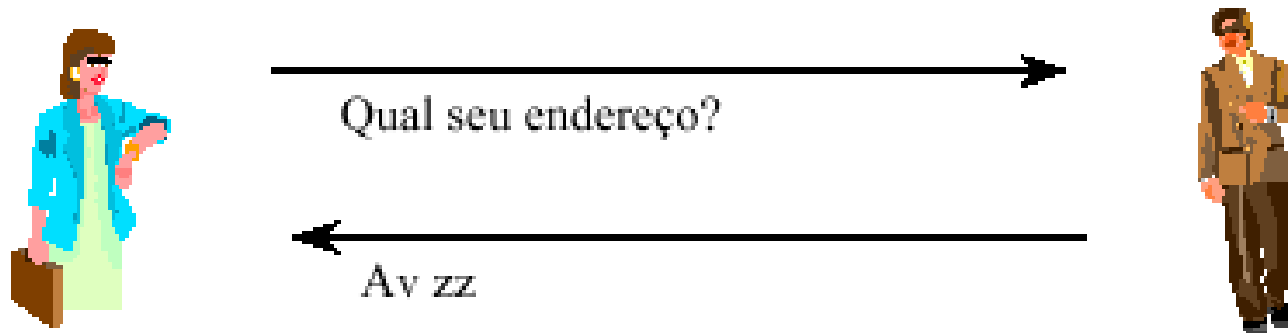
José



João

Tecnologia de Orientação a Objetos

Mensagens e Métodos: um objeto exibe algum comportamento (executa alguma operação) quando recebe um estímulo de outro objeto



Um objeto requisita a ação de algum outro objeto enviando uma Mensagem para ele.

Tecnologia de Orientação a Objetos

Uma **mensagem** contém:

- o nome do objeto receptor;
- o nome da mensagem;
- argumentos (opcional) que podem ser objetos;

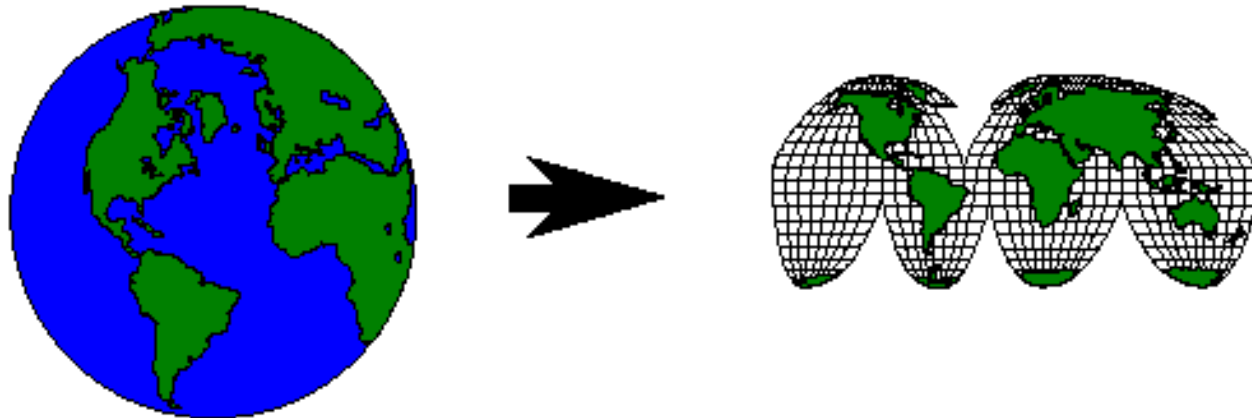
Uma **mensagem** é uma solicitação a um objeto para que seja executada uma rotina denominada **método**

Os **métodos** são responsáveis por acessar ou alterar os **atributos** de um objeto

Tecnologia de Orientação a Objetos

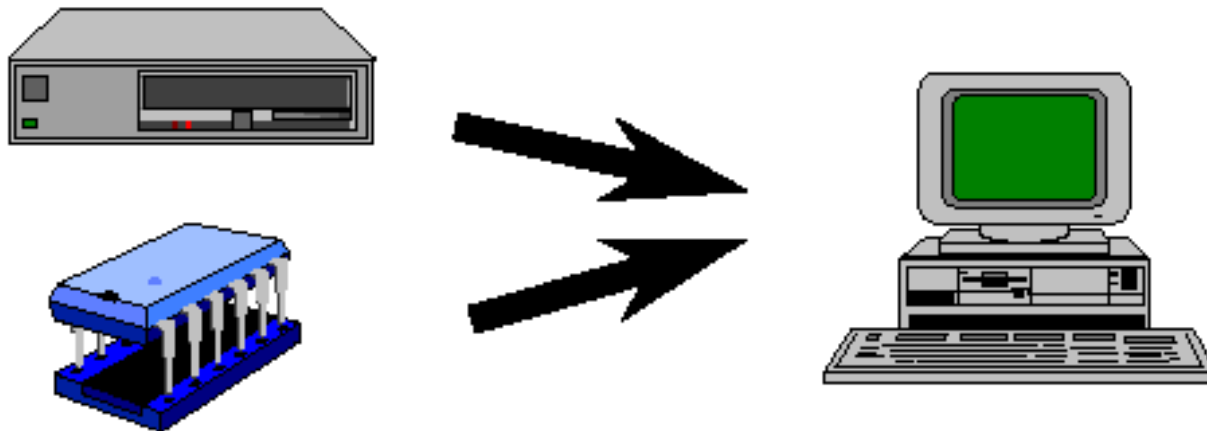
Abstração

- A OO facilita a abstração por representar mais intuitivamente objetos do mundo real.



Tecnologia de Orientação a Objetos

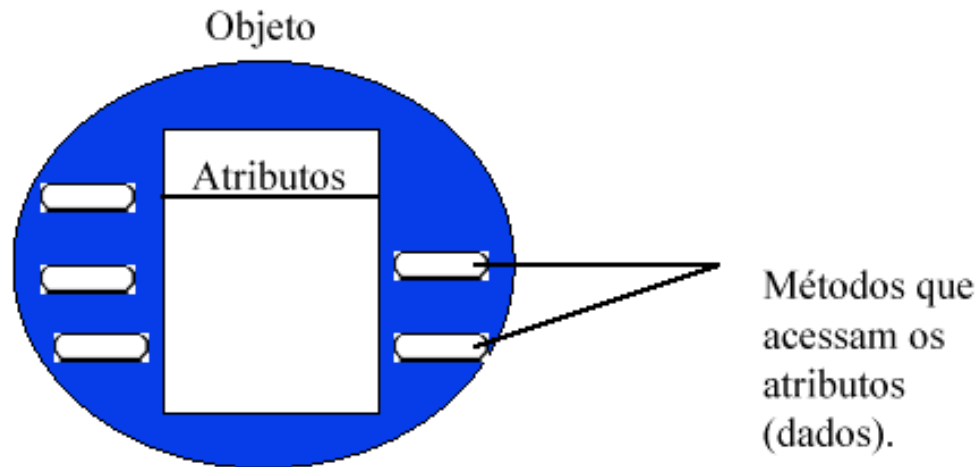
- O programador trabalha sobre componentes já desenvolvidos e testados, diminuindo a complexidade e aumentando a abstração.



Tecnologia de Orientação a Objetos

Encapsulamento:

termo formal que define o empacotamento de dados de um objeto, permitindo o acesso aos dados somente através dos métodos deste mesmo objeto.

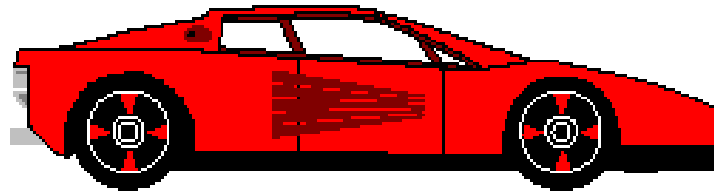


Tecnologia de Orientação a Objetos

Com o **encapsulamento**, o objeto se comporta como uma caixa-preta, aumentando a **abstração**

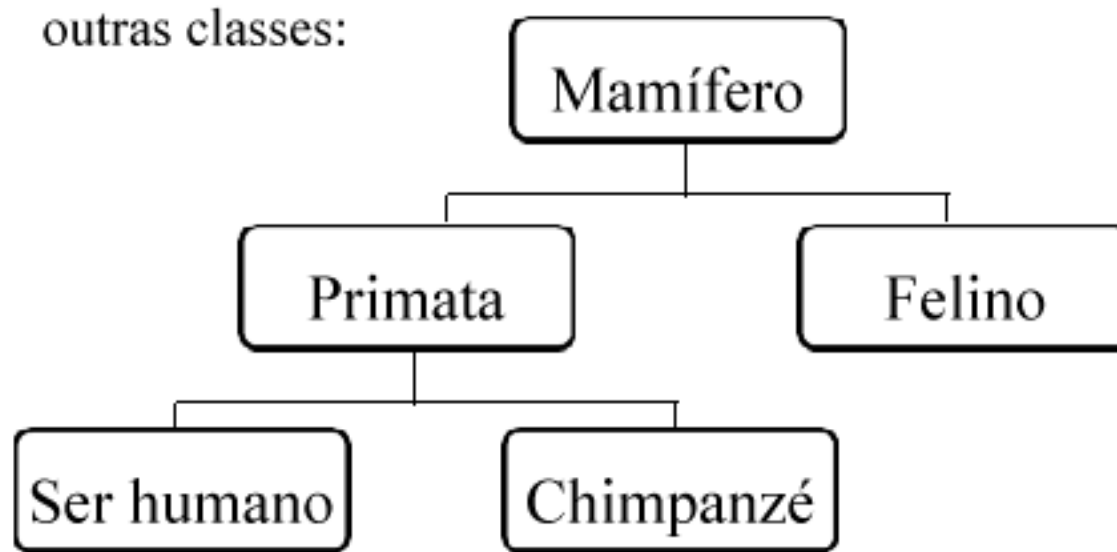


Motorista dirige um carro através dos pedais, alavanca de marchas e volante. Questões a respeito de motor estão escondidas para ele.



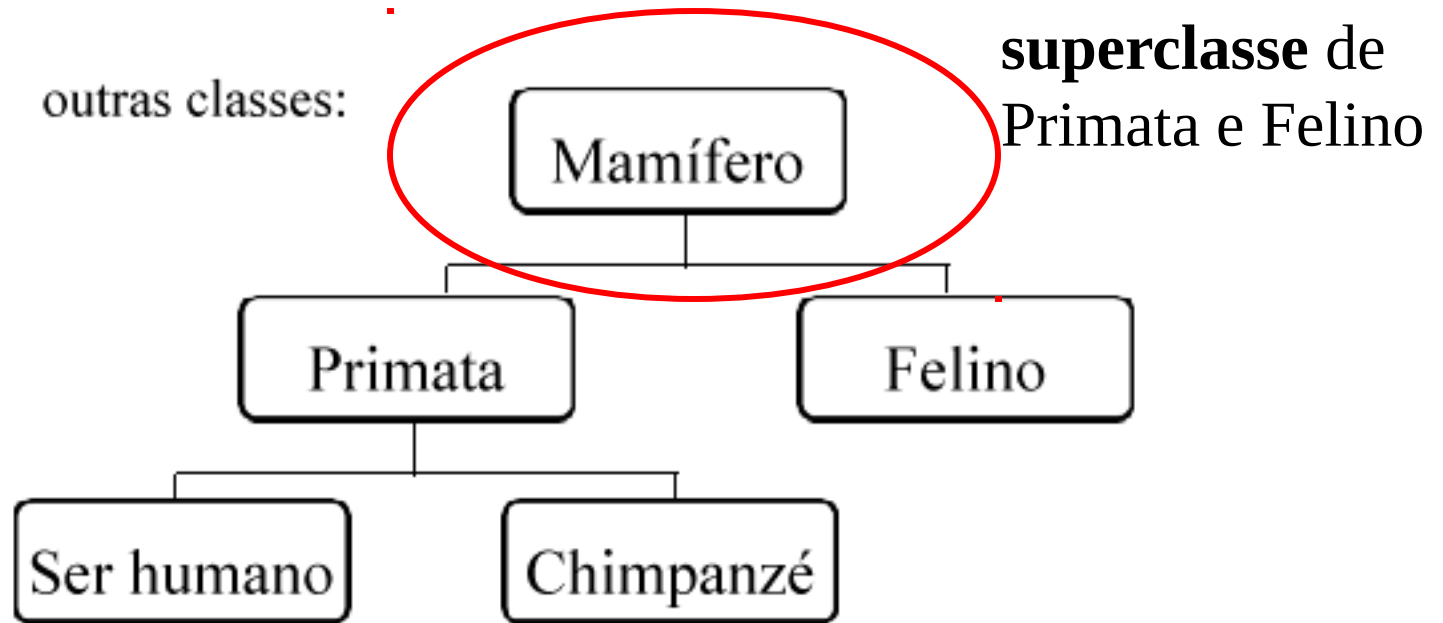
Tecnologia de Orientação a Objetos

Uma **classe** pode também resumir elementos comuns de outras classes ...



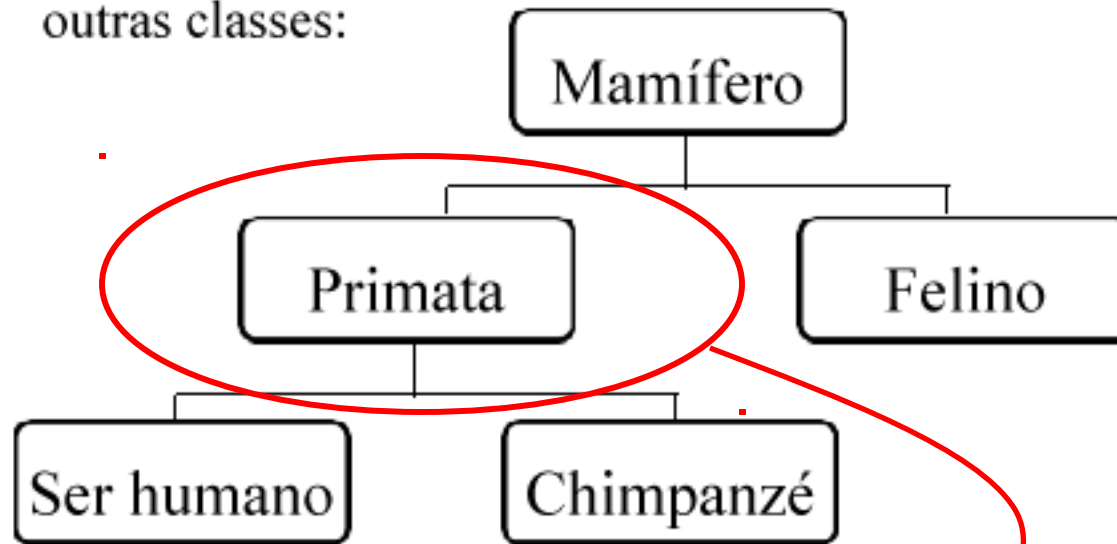
Surge então o conceitos de **subclasse** e **superclasse**.

Tecnologia de Orientação a Objetos



Conceito de Herança

outras classes:

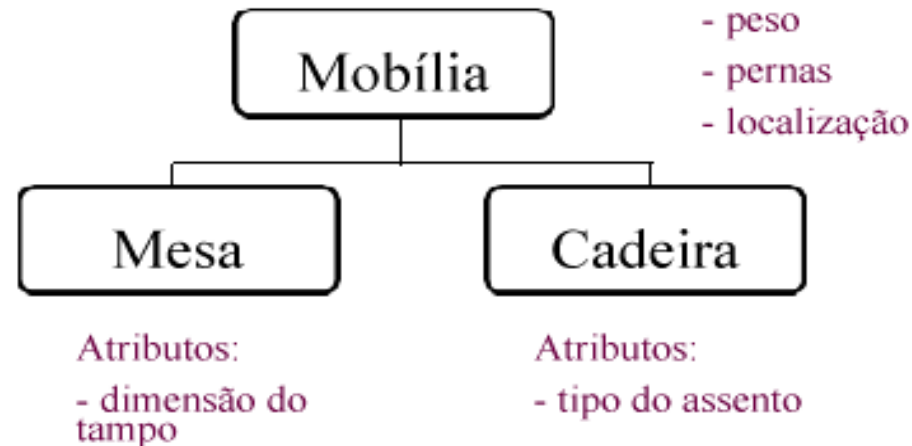


superclasse para Ser humano e Chimpanzé, as quais são subclasses de Primata

Tecnologia de Orientação a Objetos

13

Outro exemplo de herança:



Um objeto, com estado:

- marrom
- 17 kg
- 4
- sala A200
- 1 x 1,5 m

Um objeto, com estado:

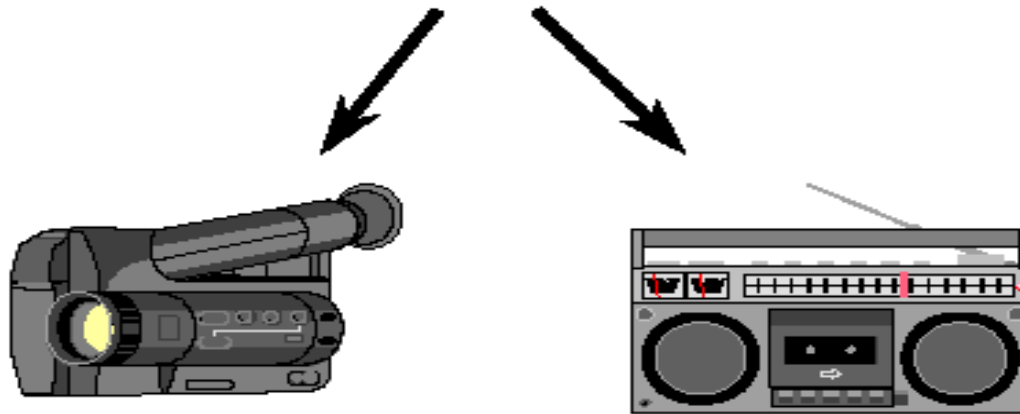
- preta
- 5 kg
- 4
- sala A200
- ondulado

Tecnologia de Orientação a Objetos

Polimorfismo

- É o nome dado à capacidade que objetos diferentes têm de responder a uma mesma mensagem.
- Mesmo nome (mensagem), formas de execução diferentes, próprias de cada objeto.

Gravar

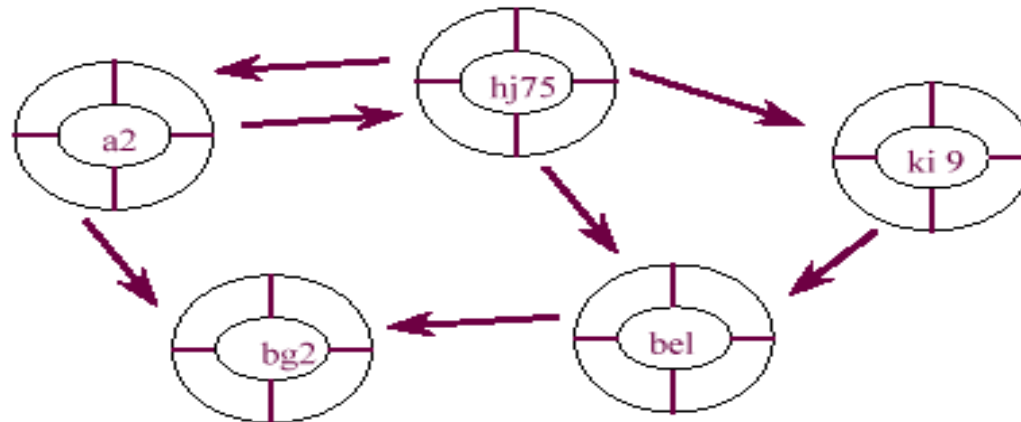


Com o polimorfismo o usuário pode enviar uma mensagem genérica e abandonar detalhes sobre a exata implementação sobre o objeto receptor.

Portanto,

O que é Orientação a Objetos ?

- É uma maneira de organizarmos o software como uma coleção de objetos discretos que incorporam :
 - uma Estrutura de Dados; e
 - um Comportamento Associado.



uma classe ...

- . contém dados
- . contém métodos
- . são reutilizáveis

Classe Contador

```
public class Contador {  
    private int num;  
    private int inicio;  
  
    public Contador(int valorInicial) {  
        inicio = valorInicial;  
        num = inicio;  
    }  
  
    public void incrementa() {  
        num = num + 1;  
    }  
  
    public int mostraNum() {  
        return num;  
    }  
}
```

um objeto ...

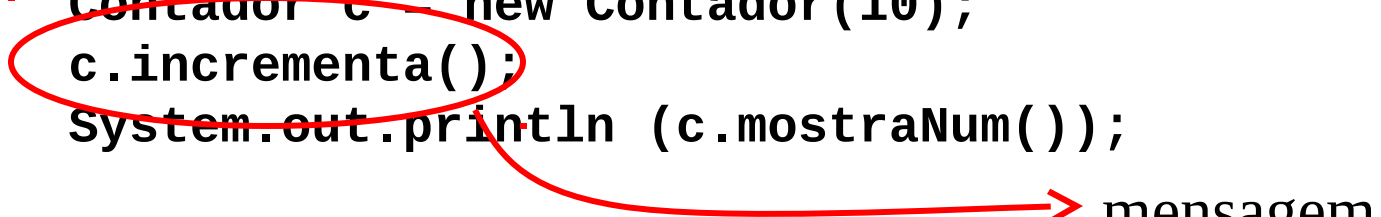
```
Contador c = new Contador(10);
```

Programa que utiliza classe Contador

```
public class TesteContador {  
    public static void main (String args[]) {  
        Contador c = new Contador(10);  
        c.incrementa();  
        System.out.println (c.mostraNum());  
  
        c.incrementa();  
        System.out.println (c.mostraNum());  
    }  
}
```

Programa que utiliza classe Contador

```
public class TesteContador {  
    public static void main (String args[]) {  
        Contador c = new Contador(10);  
        c.incrementa();  
        System.out.println (c.mostraNum());  
  
        c.incrementa();  
        System.out.println (c.mostraNum());  
    }  
}
```



mensagem

paradigmas de programação:

programação imperativa

programação orientada a objetos

paradigmas de programação:

programação imperativa

- variáveis
 - atribuições
 - loops
 - procedimentos e funções

paradigmas de programação:

programação orientada a objetos

- “tudo” é objeto
- objetos realizam computações fazendo requisições uns aos outros através de mensagens
- todo objeto possui sua própria memória
- todo objeto é instância de uma classe. Uma classe agrupa objetos similares

Tecnologia de Orientação a Objetos

Exemplo: Implementação de uma classe *Time*

Duas classes são definidas em arquivos separados:

- . Time1
- . TimeTest

Tecnologia de Orientação a Objetos

```
6 public class Time1 extends Object {
```

- definições de classe
 - classes não são normalmente criadas do início
 - use **extends** para herdar dados e métodos da classe base
 - classe derivada: classe que herda
- toda classe Java é uma subclasse de **Object**
- corpo da classe
 - delimitado por parentêses { }
 - declara variáveis de instância e métodos

Tecnologia de Orientação a Objetos

```
7   private int hour;    // 0 - 23
8   private int minute;  // 0 - 59
9   private int second;  // 0 - 59
```

– modificadores de acesso

- **public**: acessível sempre que um programa tem uma referência a um objeto da classe
- **private**: acessível somente aos métodos da classe
- normalmente as variáveis de instância são **private**

Tecnologia de Orientação a Objetos

```
21 public void setTime( int h, int m, int s )
```

```
29 public String toUniversalString()
```

– métodos

- métodos de acesso
 - métodos públicos que lêem/mostram dados
 - interface pública
 - clientes usam referências para interagir com os objetos
- métodos utilitários
 - métodos privativos que dão suporte aos métodos de acesso

Tecnologia de Orientação a Objetos

39 `public String toString()`

- método **toString**
 - da classe **Object**
 - não possui argumentos e retorna um **String**
- classes simplificam a programação
 - cliente somente precisa conhecer as operações públicas
 - cliente não é dependente dos detalhes de implementação
- reutilização de software

Tecnologia de Orientação a Objetos

14 public Time1()

- Construtor
 - método especial
 - possui o mesmo nome da classe
 - Inicializa os dados do objeto
 - garante que objetos iniciem em um estado consistente
- Construtores podem ser sobrecarregados
- Construtores não podem retornar um valor

Tecnologia de Orientação a Objetos

56 `Time1 t = new Time1(); // calls Time1 constructor`

– declarações

- uma classe pode ser usada como um tipo de dados
- declara objetos da classe
- inicializada com um construtor
- usa o operador **new** para instanciar um novo objeto

```

1 // Fig. 8.1: Time1.java
2 // Time1 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 Time1 extends Object {
7     private int hour;        // 0 - 23
8     private int minute;      // 0 - 59
9     private int second;      // 0 - 59
10
11     // Time1 constructor initializes each instance
12     // to zero. Ensures that each Time1 object has a
13     // consistent state.
14     public Time1()
15     {
16         setTime( 0, 0, 0 );
17     }
18
19     // Set a new time value using universal time. Perform
20     // validity checks on the data. Set invalid values to zero.
21     public void setTime( int h, int m, int s )
22     {
23         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26     }
27

```

Cada arquivo precisa exatamente de uma classe pública, a qual é o nome do arquivo. **Time1** herda da classe **Object**.

```

1 // Fig. 8.1: Time1.java
2 // Time1 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 Time1 extends Object {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11 // Time1 constructor initializes each
12 // to zero. Ensures that each Time1
13 // consistent state.
14 public Time1()
15 {
16     setTime( 0, 0, 0 );
17 }
18
19 // Set a new time value using univer
20 // validity checks on the data. Set
21 public void setTime( int h, int m, int s )
22 {
23     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26 }
27

```

Variáveis de instância **private** podem somente ser acessadas por métodos em suas classes.

```

1 // Fig. 8.1: Time1.java
2 // Time1 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 Time1 extends Object {
7     private int hour;        // 0 - 23
8     private int minute;      // 0 - 59
9     private int second;      // 0 - 59
10
11     // Time1 constructor initializes each instance variable
12     // to zero. Ensures that each Time1 object starts in a
13     // consistent state.
14     public Time1()
15     {
16         setTime( 0, 0, 0 );
17     }
18
19     // Set a new time value using universal
20     // validity checks on the data. Set invalid values to zero.
21     public void setTime( int h, int m, int s )
22     {
23         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26     }
27

```

Construtor **Time1**
inicializa novos
objetos **Time1**.

```

1 // Fig. 8.1: Time1.java
2 // Time1 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 Time1 extends Object {
7     private int hour;        // 0 - 23
8     private int minute;      // 0 - 59
9     private int second;      // 0 - 59
10
11     // Time1 constructor initializes each instance variable
12     // to zero. Ensures that each Time1 object starts in a
13     // consistent state.
14     public Time1()
15     {
16         setTime( 0, 0, 0 );
17     }
18
19     // Set a new time value using universal time. Perform
20     // validity checks on the data. Set invalid values to zero.
21     public void setTime( int h, int m, int s )
22     {
23         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26     }
27

```

Um método **public** pode ser acessado através de uma referência a **Time1**. Verifica a validade dos argumentos.

```

28 // Convert to String in universal-time format
29 public String toUniversalString()
30 {
31     DecimalFormat twoDigits = new DecimalFormat( "00" );
32
33     return twoDigits.format( hour ) + ":" +
34         twoDigits.format( minute ) + ":" +
35         twoDigits.format( second );
36 }
37
38 // Convert to String in standard-time format
39 public String toString()
40 {
41     DecimalFormat twoDigits = new DecimalFormat
42
43     return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
44         ":" + twoDigits.format( minute ) +
45         ":" + twoDigits.format( second ) +
46         ( hour < 12 ? " AM" : " PM" );
47 }
48 }

```

Método **toString** sabe implicitamente como usar as variáveis de instância do objeto que o invocou.

```

49 // Fig. 8.1: TimeTest.java
50 // Class TimeTest to exercise class Time1
51 import javax.swing.JOptionPane;
52
53 public class TimeTest {
54     public static void main( String args[] )
55     {
56         Time1 t = new Time1(); // calls Time1 constructor
57         String output;
58
59         output = "The initial universal time is: " +
60             t.toUniversalString() +
61             "\nThe initial standard time is: " +
62             t.toString() +
63             "\nImplicit toString() call: " + t;
64
65         t.setTime( 13, 27, 6 );
66         output += "\n\nUniversal time after setTime is: " +
67             t.toUniversalString() +
68             "\nStandard time after setTime is: " +
69             t.toString();
70
71         t.setTime( 99, 99, 99 ); // all invalid values
72         output += "\n\nAfter attempting invalid settings: " +
73             "\nUniversal time: " + t.toUniversalString() +
74             "\nStandard time: " + t.toString();

```

Classe **Time1** não precisa da declaração **import**.


```

49 // Fig. 8.1: TimeTest.java
50 // Class TimeTest to exercise class Time1
51 import javax.swing.JOptionPane;
52
53 public class TimeTest {
54     public static void main( String args[] )
55     {
56         Time1 t = new Time1(); // calls Time1 constructor
57         String output;
58
59         output = "The initial universal time is: " +
60             t.toUniversalString() +
61             "\nThe initial standard time is: " +
62             t.toString() +
63             "\nImplicit toString() call: " + t;
64
65         t.setTime( 13, 27, 6 );
66         output += "\n\nUniversal time after setTime is: " +
67             t.toUniversalString() +
68             "\nStandard time after setTime is: " +
69             t.toString();
70
71         t.setTime( 99, 99, 99 ); // all invalid values
72         output += "\n\nAfter attempting invalid settings: " +
73             "\nUniversal time: " + t.toUniversalString() +
74             "\nStandard time: " + t.toString();

```

Declara **t** como referência a um objeto **Time1**, e cria um novo objeto.

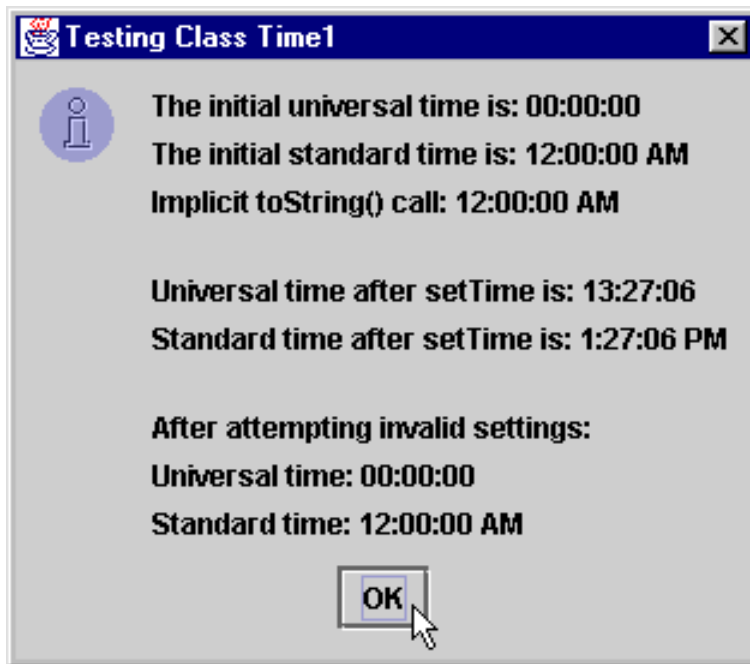
```

49 // Fig. 8.1: TimeTest.java
50 // Class TimeTest to exercise class Time1
51 import javax.swing.JOptionPane;
52
53 public class TimeTest {
54     public static void main( String args[] )
55     {
56         Time1 t = new Time1(); // calls Time1 constructor
57         String output;
58
59         output = "The initial universal time is: " +
60             t.toUniversalString() +
61             "\nThe initial standard time is: " +
62             t.toString() +
63             "\nImplicit toString() call\n";
64
65         t.setTime( 13, 27, 6 );
66         output += "\n\nUniversal time after setTime is: " +
67             t.toUniversalString() +
68             "\nStandard time after setTime is: " +
69             t.toString();
70
71         t.setTime( 99, 99, 99 ); // all invalid values
72         output += "\n\nAfter attempting invalid settings: " +
73             "\nUniversal time: " + t.toUniversalString() +
74             "\nStandard time: " + t.toString();

```

Chama método **setTime** usando a referência **t**. Note o ponto.

```
75
76     JOptionPane.showMessageDialog( null, output,
77         "Testing Class Time1",
78         JOptionPane.INFORMATION_MESSAGE );
79
80     System.exit( 0 );
81 }
82 }
```



- escopo de classe
 - variáveis de instância e métodos
 - variáveis de instância são acessíveis através dos métodos
 - podem ser referenciadas pelo nome
 - Membros públicos (**public**) são acessados através de uma referência
 - referenciaObjeto.nomeVariavel**
 - referenciaObjeto.nomeMetodo()**

- Propósito do **public**
 - Oferece aos clientes uma visão dos *serviços* que a classe fornece (interface)
- Propósito do **private**
 - Oculta detalhes de implementação
 - membros **Private** somente são acessíveis através da interface pública (**public**) usando métodos públicos

```
1 // Fig. 8.2: TimeTest.java
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 public class TimeTest {
5     public static void main( String args[] )
6     {
7         Time1 t = new Time1();
8
9         t.hour = 7;
10    }
11 }
```

Tentativa de acessar uma variável de instância **private** da classe **Time1**.

TimeTest.java:9: Variable hour in class Time1 not
accessible from class TimeTest.

```
t.hour = 7;
  ^
```

1 error

- Pacotes
 - Estruturas de diretório que organizam classes e interfaces
 - Mecanismo para reutilização de software
- Criando pacotes
 - Crie uma classe **public**
 - Se não é **public**, pode ser acessada apenas por classes no mesmo pacote
 - Escolha um nome para o pacote e adicione a declaração **package** ao código fonte
 - Compile a classe (deve ser colocada no diretório apropriado)

- Construtor
 - Pode inicializar membros do objeto
 - não há retorno
 - Classe pode ter construtores sobrecarregados
 - Inicializadores são passados como argumentos ao construtor
 - parâmetros no construtor são opcionais
 - Declaração/inicialização de novos objetos:
ref = new NomeClasse(argumentos);
 - Construtor tem o mesmo nome da classe

- Construtor
 - Se nenhum construtor está definido, o compilador define um construtor padrão
 - Defaults: **0** para tipos numéricos primitivos, **false** para **boolean**, **null** para referências

Tecnologia de Orientação a Objetos

- Exemplo

Vários construtores são definidos

```
15    public Time2() { setTime( 0, 0, 0 ); }  
19    public Time2( int h ) { setTime( h, 0, 0 ); }  
23    public Time2( int h, int m ) { setTime( h, m, 0 ); }  
26    public Time2( int h, int m, int s ) { setTime( h, m, s ); }
```

– Nota:

- Quando um objeto tem uma referência a outro objeto da mesma classe, este pode acessar todos os dados e métodos

```

1 // Fig. 8.4: Time2.java
2 // Time2 class definition
3 package com.deitel.jhtp3.ch08;    // place Time2 in a package
4 import java.text.DecimalFormat;  // used for number formatting
5
6 // This class maintains the time
7 public class Time2 extends Object
8     private int hour;           // 0
9     private int minute;        // 0 - 59
10    private int second;         // 0 - 59
11
12    // Time2 constructor initializes each instance variable
13    // to zero. Ensures that Time object starts in a
14    // consistent state.
15    public Time2() { setTime( 0, 0, 0 ); }
16
17    // Time2 constructor: hour supplied, minute and second
18    // defaulted to 0.
19    public Time2( int h ) { setTime( h, 0, 0 ); }
20
21    // Time2 constructor: hour and minute supplied, second
22    // defaulted to 0.
23    public Time2( int h, int m ) { setTime( h, m, 0 ); }
24
25    // Time2 constructor: hour, minute and second supplied.
26    public Time2( int h, int m, int s ) { setTime( h, m, s ); }
27

```

Classe **Time2** é colocada dentro da package

```

1 // Fig. 8.4: Time2.java
2 // Time2 class definition
3 package com.deitel.jhtp3.ch08;    // place Time2 in a package
4 import java.text.DecimalFormat;  // used for number formatting
5
6 // This class maintains the time in 24-hour format
7 public class Time2 extends Object {
8     private int hour;           // 0 - 23
9     private int minute;        // 0 - 59
10    private int second;         // 0 - 59
11
12    // Time2 constructor initializes each instance
13    // to zero. Ensures that Time object starts in
14    // consistent state.
15    public Time2() { setTime( 0, 0, 0 ); }
16
17    // Time2 constructor: hour supplied, minute and
18    // defaulted to 0.
19    public Time2( int h ) { setTime( h, 0, 0 ); }
20
21    // Time2 constructor: hour and minute supplied, second
22    // defaulted to 0.
23    public Time2( int h, int m ) { setTime( h, m, 0 ); }
24
25    // Time2 constructor: hour, minute and second supplied.
26    public Time2( int h, int m, int s ) { setTime( h, m, s ); }
27

```

Note os construtores sobrecarregados. O construtor apropriado é invocado conforme o número de argumentos.

```

28 // Time2 constructor: another Time2 object supplied.
29 public Time2( Time2 time )
30 {
31     setTime( time.hour, time.minute, time.second );
32 }
33
34 // Set a new time value using universal time
35 // validity checks on the data. Set invalid
36 public void setTime( int h, int m, int s )
37 {
38     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
39     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
40     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
41 }
42
43 // Convert to String in universal-time format
44 public String toUniversalString()
45 {
46     DecimalFormat twoDigits = new DecimalFormat( "00" );
47
48     return twoDigits.format( hour ) + ":" +
49         twoDigits.format( minute ) + ":" +
50         twoDigits.format( second );
51 }
52
53 // Convert to String in standard-time format
54 public String toString()
55 {
56     DecimalFormat twoDigits = new DecimalFormat( "00" );
57

```


Construtor recebe um objeto **Time2**. O construtor pode acessar todos os dados no objeto **Time2** porque eles são objetos da mesma classe.

```

64 // Fig. 8.4: TimeTest.java
65 // Using overloaded constructors
66 import javax.swing.*;
67 import com.deitel.jhttp3.ch08.Time2;
68
69 public class TimeTest {
70     public static void main( String args[] )
71     {
72         Time2 t1, t2, t3, t4, t5, t6;
73         String output;
74
75         t1 = new Time2();
76         t2 = new Time2( 2 );
77         t3 = new Time2( 21, 34 );
78         t4 = new Time2( 12, 25, 42 );
79         t5 = new Time2( 27, 74, 99 );
80         t6 = new Time2( t4 );    // use t4 as initial value
81
82         output = "Constructed with: " +
83                 "\nt1: all arguments defaulted" +
84                 "\n      " + t1.toUniversalString() +
85                 "\n      " + t1.toString();
86
87         output += "\nt2: hour specified; minute and " +
88                 "second defaulted" +
89                 "\n      " + t2.toUniversalString() +
90                 "\n      " + t2.toString();
91

```

Inicializa objetos **Time2**
usando os construtores
sobrecarregados



- Métodos *Set*
 - Método público (**public**) que dá valores a variáveis **private**
 - Não viola a noção de dados **private**
 - Permite a mudança de variáveis que você quer
 - Métodos que alteram valor das variáveis
- Métodos *Get*
 - Método público (**public**) que mostra variáveis **private**
 - De novo, não viola a noção de dados **private**
 - Somente mostra a informação que você quer mostrar

```
1 // Fig. 8.5: Time3.java
2 // Time3 class definition
3 package com.deitel.jhtp3.ch08;    // place Time3 in a package
4 import java.text.DecimalFormat;  // used for number formatting
5
6 // This class maintains the time in 24-hour format
7 public class Time3 extends Object {
8     private int hour;           // 0 - 23
9     private int minute;        // 0 - 59
10    private int second;         // 0 - 59
11
12    // Time3 constructor initializes each instance variable
13    // to zero. Ensures that Time object starts in a
14    // consistent state.
15    public Time3() { setTime( 0, 0, 0 ); }
16
17    // Time3 constructor: hour supplied, minute and second
18    // defaulted to 0.
19    public Time3( int h ) { setTime( h, 0, 0 ); }
20
21    // Time3 constructor: hour and minute supplied, second
22    // defaulted to 0.
23    public Time3( int h, int m ) { setTime( h, m, 0 ); }
24
25    // Time3 constructor: hour, minute and second supplied.
26    public Time3( int h, int m, int s ) { setTime( h, m, s ); }
27
```



```

28 // Time3 constructor: another Time3 object supplied.
29 public Time3( Time3 time )
30 {
31     setTime( time.getHour(),
32             time.getMinute(),
33             time.getSecond() );
34 }
35
36 // Set Methods
37 // Set a new time value using universal time. Perform
38 // validity checks on the data. Set invalid values to zero.
39 public void setTime( int h, int m, int s )
40 {
41     setHour( h );    // set the hour
42     setMinute( m );  // set the minute
43     setSecond( s );  // set the second
44 }
45
46 // set the hour
47 public void setHour( int h )
48     { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }
49
50 // set the minute
51 public void setMinute( int m )
52     { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }
53
54 // set the second
55 public void setSecond( int s )
56     { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }
57

```

Os métodos públicos *set* permitem que os clientes mudem dados **private**. Entretanto as entradas são validadas.

```

58 // Get Methods
59 // get the hour
60 public int getHour() { return hour; }
61
62 // get the minute
63 public int getMinute() { return minute; }
64
65 // get the second
66 public int getSecond() { return second; }
67
68 // Convert to String in universal-time format
69 public String toUniversalString()
70 {
71     DecimalFormat twoDqaits = new DecimalFormat( "00" );
72
73     return twoDqaits.format( getHour() ) + ":" +
74         twoDqaits.format( getMinute() ) + ":" +
75         twoDqaits.format( getSecond() );
76 }
77
78 // Convert to String in standard-time format
79 public String toString()
80 {
81     DecimalFormat twoDqaits = new DecimalFormat( "00" );
82
83     return ( ( getHour() == 12 || getHour() == 0 ) ?
84         12 : getHour() % 12 ) + ":" +
85         twoDqaits.format( getMinute() ) + ":" +
86         twoDqaits.format( getSecond() ) +
87         ( getHour() < 12 ? " AM" : " PM" );
88 }
89 }

```

Métodos *get* permitem que os clientes visualizem os dados **private** selecionados.

- Composição
 - Classe tem referências a outros objetos como membros
 - não é obrigatório que seja imediatamente inicializado
 - Construtor default é automaticamente chamado
 - Exemplo:
objeto **AlarmClock** tem um objeto **Time** como um membro

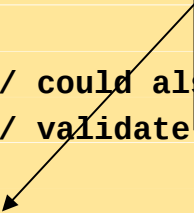
- Exemplo
 - Classe **Date**
 - variáveis de instância **month, day, year**
 - Classe **Employee**
 - Variáveis de instância **firstName, lastName**
 - Composition: tem as referências de **Date** **birthDate** e **hireDate**

```

1 // Fig. 8.8: Date.java
2 // Declaration of the Date class.
3 package com.deitel.jhttp3.ch08;
4
5 public class Date extends Object {
6     private int month; // 1-12
7     private int day;    // 1-31 based on month
8     private int year;   // any year
9
10    // Constructor: Confirm proper value for month;
11    // call method checkDay to confirm proper
12    // value for day.
13    public Date( int mn, int dy, int yr )
14    {
15        if ( mn > 0 && mn <= 12 ) // validate the month
16            month = mn;
17        else {
18            month = 1;
19            System.out.println( "Month " + mn +
20                               " invalid. Set to month 1." );
21        }
22
23        year = yr; // could al
24        day = checkDay( dy ); // validate the day
25
26        System.out.println(
27            "Date object constructor for date " + toString() );
28    }
29

```

**Construtor imprime
quando invocado**



```

30 // Utility method to confirm proper day value
31 // based on month and year.
32 private int checkDay( int testDay )
33 {
34     int daysPerMonth[] = { 0, 31, 28, 31, 30,
35                             31, 30, 31, 31, 30,
36                             31, 30, 31 };
37
38     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39         return testDay;
40
41     if ( month == 2 && // February: Check for leap year
42         testDay == 29 &&
43         ( year % 400 == 0 ||
44           ( year % 4 == 0 && year % 100 != 0 ) ) )
45         return testDay;
46
47     System.out.println( "Day " + testDay +
48                         " invalid. Set to 1." );
49
50     return 1; // leave object in consistent state
51 }
52
53 // Create a String of the form month/day/year
54 public String toString()
55 { return month + "/" + day + "/" + year; }
56 }

```

← Checa a validade da data.

← Sobre põe o método **toString** (da classe **Object**).

```

57 // Fig. 8.8: Employee.java
58 // Declaration of the Employee class.
59 package com.deitel.jhttp3.ch08;
60
61 public class Employee extends Object {
62     private String firstName;
63     private String lastName;
64     private Date birthDate;
65     private Date hireDate;
66
67     public Employee( String fName, String lName,
68                     int bMonth, int bDay, int bYear,
69                     int hMonth, int hDay, int hYear)
70     {
71         firstName = fName;
72         lastName = lName;
73         birthDate = new Date( bMonth, bDay, bYear );
74         hireDate = new Date( hMonth, hDay, hYear );
75     }
76
77     public String toString()
78     {
79         return lastName + ", " + firstName +
80             "   Hired: " + hireDate.toString() +
81             "   Birthday: " + birthDate.toString();
82     }
83 }

```

Composição:
referências **Date**.

Construtor
Employee instancia
novos objetos **Date**.

```

84 // Fig. 8.8: EmployeeTest.java
85 // Demonstrating an object with a member object.
86 import javax.swing.JOptionPane;
87 import com.deitel.jhtp3.ch08.Employee;
88
89 public class EmployeeTest {
90     public static void main( String args[] )
91     {
92         Employee e = new Employee( "Bob", "Jones", 7, 24, 49,
93                                   3, 12, 88 );
94         JOptionPane.showMessageDialog( null, e.toString(),
95                                     "Testing Class Employee",
96                                     JOptionPane.INFORMATION_MESSAGE );
97
98         System.exit( 0 );
99     }
100 }

```

Constructor **Employee**
invoca o construtor **Date**.



Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988

- Cada objeto tem uma referência a si mesmo
 - A referência **this**
 - Usado para referenciar variáveis de instância e métodos
- Dentro dos métodos
 - Se o parâmetro tem o mesmo nome das variáveis de instância
 - variável de instância oculta
 - Use **this.nomeVariavel** para referenciar explicitamente a variável de instância

- Dentro dos métodos (cont.)
 - Use **nomeVariavel** para referenciar o parâmetro
 - Melhora a legibilidade do código

- Exemplo de encadeamento
 - os métodos **setHour**, **setMinute**, e **setSecond** retornam **this**
 - Para o objeto **t**, considere
t.setHour(1).setMinute(2).setSecond(3);
 - Executa **t.setHour(1)** e retorna **this**, então a expressão se torna
t.setMinute(2).setSecond(3);
 - Executa **t.setMinute(2)**, retorna a referência, e se torna
t.setSecond(3);
 - Executa **t.setSecond(3)**, retorna a referência, e se torna **t**

```

1 // Fig. 8.11: Time4.java
2 // Time4 class definition
3 package com.deitel.jhtp3.ch08;    // place Time4 in a package
4 import java.text.DecimalFormat;  // used for number formatting
5
6 // This class maintains the time in 24-hour format
7 public class Time4 extends Object {
8     private int hour;           // 0 - 23
9     private int minute;        // 0 - 59
10    private int second;         // 0 - 59
11
12    // Time4 constructor initializes each instance variable
13    // to zero. Ensures that Time object starts in a
14    // consistent state.
15    public Time4() { this.setTime( 0, 0, 0 ); }
16
17    // Time4 constructor: hour supplied, minute and second
18    // defaulted to 0.
19    public Time4( int h ) { this.setTime( h, 0, 0 ); }
20
21    // Time4 constructor: hour and minute supplied, second
22    // defaulted to 0.
23    public Time4( int h, int m ) { this.setTime( h, m, 0 ); }
24
25    // Time4 constructor: hour, minute and second supplied.
26    public Time4( int h, int m, int s )
27        { this.setTime( h, m, s ); }
28

```


Note o uso do **this** no construtor.

```

29 // Time4 constructor: another Time4 object supplied.
30 public Time4( Time4 time )
31 {
32     this.setTime( time.getHour(),
33                  time.getMinute(),
34                  time.getSecond() );
35 }
36
37 // Set Methods
38 // Set a new Time value using military time. Perform
39 // validity checks on the data. Set invalid values to zero.
40 public Time4 setTime( int h, int m, int s )
41 {
42     this.setHour( h );    // set the hour
43     this.setMinute( m ); // set the minute
44     this.setSecond( s ); // set the second
45
46     return this;    // enables chaining
47 }
48
49 // set the hour
50 public Time4 setHour( int h )
51 {
52     this.hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
53
54     return this;    // enables chaining
55 }
56

```

Retornando a referência **this** habilita o encadeamento.



```
57 // set the minute
58 public Time4 setMinute( int m )
59 {
60     this.minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
61
62     return this;    // enables chaining
63 }
64
65 // set the second
66 public Time4 setSecond( int s )
67 {
68     this.second = ( ( s >= 0 && s < 60 ) ? s : 0 );
69
70     return this;    // enables chaining
71 }
72
73 // Get Methods
74 // get the hour
75 public int getHour() { return this.hour; }
76
77 // get the minute
78 public int getMinute() { return this.minute; }
79
80 // get the second
81 public int getSecond() { return this.second; }
82
```

```

83 // Convert to String in universal-time format
84 public String toUniversalString()
85 {
86     DecimalFormat twoDigits = new DecimalFormat( "00" );
87
88     return twoDigits.format( this.getHour() ) + ":" +
89         twoDigits.format( this.getMinute() ) + ":" +
90         twoDigits.format( this.getSecond() );
91 }
92
93 // Convert to String in standard-time format
94 public String toString()
95 {
96     DecimalFormat twoDigits = new DecimalFormat( "00" );
97
98     return ( ( this.getHour() == 12 ||
99         this.getHour() == 0 ) ?
100         12 : this.getHour() % 12 ) + ":" +
101         twoDigits.format( this.getMinute() ) + ":" +
102         twoDigits.format( this.getSecond() ) +
103         ( this.getHour() < 12 ? " AM" : " PM" );
104 }
105}

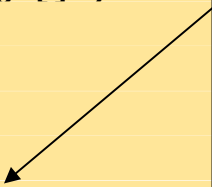
```

```

106// Fig. 8.11: TimeTest.java
107// Chaining method calls together with the this reference
108import javax.swing.*;
109import com.deitel.jhttp3.ch08.Time4;
110
111public class TimeTest {
112    public static void main( String args[] )
113    {
114        Time4 t = new Time4();
115        String output;
116
117        t.setHour( 18 ).setMinute( 30 ).setSecond(
118
119        output = "Universal time: " + t.toUniversal() +
120                "\nStandard time: " + t.toString() +
121                "\n\nNew standard time: " +
122                t.setTime( 20, 20, 20 ).toString();
123
124        JOptionPane.showMessageDialog( null, output,
125            "Chaining Method Calls",
126            JOptionPane.INFORMATION_MESSAGE );
127
128        System.exit( 0 );
129    }
130}

```

Retornando a referência **this** permite invocação a métodos em cascata.



- Memória
 - Construtores usam memória quando novos objetos são criados
 - coleta de lixo automática (*garbage collection*)
 - Quando objeto não é mais usado é marcado para a coleta de lixo
 - Coleta de lixo restaura a memória
 - Falhas quanto à memória é menos comum em Java que em C e C++
- método **finalizer**
 - devolve os recursos para o sistema
 - não tem parâmetros, não retorna valor

- Variáveis estáticas
 - palavra-chave **static**
 - usualmente cada objeto possui uma cópia de cada variável
 - variáveis de classe estáticas são compartilhadas entre todos os objetos da classe
 - Uma cópia para a classe inteira usar
 - Possui escopo de classe (não é global)

- Variáveis estáticas
 - membros **public static**
 - Acessados através do nome da classe e operador ponto
MinhaClasse.minhaVariavelEstatica
 - membros **private static**
 - Acessados através de métodos
 - Se nenhum objeto existe, o nome da classe e o método **public static** deve ser usado
MinhaClasse.meuMetodo()

- **Métodos estáticos (static)**
 - Somente pode acessar membros **static**
 - Não possui referência **this**
 - variáveis **static** são independentes de objetos

```

1 // Fig. 8.12: Employee.java
2 // Declaration of the Employee class.
3 public class Employee extends Object {
4     private String firstName;
5     private String lastName;
6     private static int count; // # of objects in memory
7
8     public Employee( String fName, String lName )
9     {
10         firstName = fName;
11         lastName = lName;
12
13         ++count; // increment static count of employees
14         System.out.println( "Employee object finalizer: " +
15                             firstName + " " + lastName );
16     }
17
18     protected void finalize()
19     {
20         --count; // decrement static count of employees
21         System.out.println( "Employee object finalizer: " +
22                             firstName + " " + lastName +
23                             "; count = " + count );
24     }
25
26     public String getFirstName() { return firstName; }
27
28     public String getLastName() { return lastName; }
29
30     public static int getCount() { return count; }
31 }

```

count declarado **static** e **private**. Ele deve ser acessado através de método **static**.

```

1 // Fig. 8.12: Employee.java
2 // Declaration of the Employee class.
3 public class Employee extends Object {
4     private String firstName;
5     private String lastName;
6     private static int count; // # of objects in memory
7
8     public Employee( String fName, String lName )
9     {
10         firstName = fName;
11         lastName = lName;
12
13         ++count; // increment static count of employees
14         System.out.println( "Employee object constructor: " +
15                             firstName + " " + lastName );
16     }
17
18     protected void finalize()
19     {
20         --count; // decrement static count of employees
21         System.out.println( "Employee object finalize: " +
22                             firstName + " " + lastName +
23                             "; count = " + count );
24     }
25
26     public String getFirstName() { return firstName; }
27
28     public String getLastName() { return lastName; }
29
30     public static int getCount() { return count; }
31 }

```

Construtor e finalizador incrementa/decrementa **count** quando objetos **Employee** são criados.

```

1 // Fig. 8.12: Employee.java
2 // Declaration of the Employee class.
3 public class Employee extends Object {
4     private String firstName;
5     private String lastName;
6     private static int count; // # of objects in memory
7
8     public Employee( String fName, String lName )
9     {
10         firstName = fName;
11         lastName = lName;
12
13         ++count; // increment static count of employees
14         System.out.println( "Employee object constructor: " +
15                             firstName + " " + lastName );
16     }
17
18     protected void finalize()
19     {
20         --count; // decrement static count of employees
21         System.out.println( "Employee object finalizer: " +
22                             firstName + " " + lastName +
23                             "; count = " + count );
24     }
25
26     public String getFirstName() { return firstName; }
27
28     public String getLastName() { return lastName; }
29
30     public static int getCount() { return count; }
31 }

```

O método **public static getCount** permite que o **count** seja acessado, até mesmo se nenhum objeto **Employee** existir.

```

32 // Fig. 8.12: EmployeeTest.java
33 // Test Employee class with static class variable,
34 // static class method, and dynamic memory.
35 import javax.swing.*;
36
37 public class EmployeeTest {
38     public static void main( String args[] )
39     {
40         String output;
41
42         output = "Employees before instantiation: " +
43             Employee.getCount();
44
45         Employee e1 = new Employee( "Susan", "Baker" );
46         Employee e2 = new Employee( "Bob", "Jones" );
47
48         output += "\n\nEmployees after instantiation: " +
49             "\nvia e1.getCount(): " + e1.getCount() +
50             "\nvia e2.getCount(): " + e2.getCount() +
51             "\nvia Employee.getCount(): " +
52             Employee.getCount();
53
54         output += "\n\nEmployee 1: " + e1.getFirstName() +
55             " " + e1.getLastName() +
56             "\nEmployee 2: " + e2.getFirstName() +
57             " " + e2.getLastName();
58

```

O método **getCount** pode ser invocado, mesmo se nenhum objeto da classe **Employee** existe.


```

32 // Fig. 8.12: EmployeeTest.java
33 // Test Employee class with static class variable,
34 // static class method, and dynamic memory.
35 import javax.swing.*;
36
37 public class EmployeeTest {
38     public static void main( String args[] )
39     {
40         String output;
41
42         output = "Employees before instantiation: " +
43             Employee.getCount();
44
45         Employee e1 = new Employee( "Susan", "Baker" );
46         Employee e2 = new Employee( "Bob", "Jones" );
47
48         output += "\n\nEmployees after instantiation: " +
49             "\nvia e1.getCount(): " + e1.getCount() +
50             "\nvia e2.getCount(): " + e2.getCount() +
51             "\nvia Employee.getCount(): " +
52             Employee.getCount();
53
54         output += "\n\nEmployee 1: " + e1.getFirstName() +
55             " " + e1.getLastName() +
56             "\nEmployee 2: " + e2.getFirstName() +
57             " " + e2.getLastName();
58

```

Invoca **getCount**
usando referências de
Employee.

```
59 // mark objects referred to by e1 and e2
60 // for garbage collection
61 e1 = null;
62 e2 = null;
63
64 System.gc(); // suggest that garbage collector be called
65
66 output += "\n\nEmployees after System.gc(): " +
67           Employee.getCount();
68
69 JOptionPane.showMessageDialog( null, output,
70                                "Static Members and Garbage Collection",
71                                JOptionPane.INFORMATION_MESSAGE );
72 System.exit( 0 );
73 }
74 }
```

Uma vez que os objetos foram destruídos, a invocação a **getCount** deve ser feita através da classe.

- Ocultamento da Informação (*Information hidden*)
 - Classes ocultam de seus clientes os detalhes da implementação
 - Abstração de Dados
 - Código não dependem dos detalhes da implementação

Tecnologia de Orientação a Objetos

- Java
 - Foco na criação de tipos de dados (classes)
- Abstract Data Types (ADTs)
 - Forma de representar noções do mundo real
 - Imperfeito
 - **float** possui precisão finita
 - **int** possui um tamanho máximo
 - Captura duas noções
 - Representação dos dados e Operações
 - Usa classes para implementá-los
 - Estende a linguagem de programação base