

Linguagem Java

Adenilso da Silva Simão

15 de setembro de 2005

Definindo Classes e Subclasses

- ▶ Métodos podem ser sobrecarregados
 - ▶ Mesmo nome
 - ▶ Parâmetros distintos
 - ▶ O tipo de retorno não é suficiente para distinguir os métodos
- ▶ Não é polimorfismo
 - ▶ Em tempo de compilação é possível saber a qual dos métodos sobrecarregados a invocação se refere
- ▶ Deve ser usado apenas quando a semântica for mantida

Definindo Classes e Subclasses (II)

- Três métodos sobrecarregados f

```
1 class UmaClasse {  
2     void f() {  
3     }  
4     // Soh mudei o tipo de retorno!  
5     // int f() {  
6     // }  
7     int f(int x) {  
8         return 0;  
9     }  
10    int f(int x, float y){  
11        return 0;  
12    }
```

Definindo Classes e Subclasses (III)

- ▶ Uma classe pode definir um ou mais construtores
 - ▶ Um construtor tem o mesmo nome da classe
 - ▶ Um construtor não é um método
 - ▶ Não é herdado
 - ▶ Não tem retorno

```
1 class UmaClasse {  
2     public UmaClasse(int x) {  
3     }  
4 }
```

Criando uma Classe que não Pode Ser Instanciada

- ▶ Se nenhum construtor é definido
 - ▶ Usa-se o construtor *default*
 - ▶ Um construtor sem parâmetro
- ▶ Se ao menos um construtor é definido
 - ▶ Não se pode usar o construtor *default*
- ▶ Se uma classe não tem acesso aos construtores de outra classe, não pode criar instâncias
 - ▶ E se o construtor for `private`
 - ▶ Ninguém consegue criar objetos ☺
 - ▶ Qual a utilidade disso?

Criando um *Singleton*

- Algumas a classe pode controlar quais e quantos objetos são criados
 - Imagine que deva existir apenas um objeto de uma classe

```
1 class UmaClasse {  
2     private UmaClasse() { }  
3     static UmaClasse unica = null;  
4     static public UmaClasse facaUmaClasse() {  
5         if (unica == null) {  
6             unica = new UmaClasse();  
7         }  
8         return unica;  
9     }  
10 }
```

Criando um *Singleton* (II)

- Como conseguir um objeto *UmaClasse*?

```
1 class OutraClasse {  
2     void f() {  
3         //      UmaClasse x = new UmaClasse();  
4         UmaClasse y = UmaClasse.facaUmaClasse();  
5     }  
6 }
```

this e super

- ▶ Dentro de um método, pode-se utilizar o a palavra reservada `this` para referir-se ao objeto que está recebendo a mensagem
 - ▶ Semelhante ao `self` de *Smalltalk*
- ▶ Útil
 - ▶ para evitar ambigüidades
 - ▶ Um objeto passar a si mesmo como parâmetro

```
1 class UmaClasse {  
2     int x;  
3     void h(int x) {  
4         this.x = x;  
5     }  
6     void f(OutraClasse f) {  
7     }  
8 }
```

this e super (II)

► Passando-se por parâmetro

```
1 class OutraClasse {  
2     void g() {  
3         UmaClasse x = new UmaClasse();  
4         x.f(this);  
5     }  
6 }
```

this e super (III)

- ▶ A palavra reservada `super` permite referir-se a superclasse referente ao objeto

```
1 class UmaClasse {  
2     UmaClasse(int x) { }  
3 }  
4  
5 class OutraClasse extends UmaClasse {  
6     OutraClasse() {  
7         // UmaClasse(1); Criaria outro objeto  
8         super(1);  
9     }  
10 }
```

Herança

- ▶ Toda classe java possui uma superclasse
 - ▶ Se nenhuma superclasse é explicitamente declarada, sua superclasse será `Object`
- ▶ Não há herança múltipla
 - ▶ Mas há interfaces ☺

```
1 class UmaClasse { // superclasse é Object
2   ...
3 }
4 class OutraClasse extends UmaClasse { // superclasse é
5                                           // UmaClasse
6   ...
7 }
```

Herança (II)

- ▶ A subclasse herda todos os membros de sua superclasse
 - ▶ Atributos
 - ▶ Métodos
 - ▶ Mas não herda construtores

```
1 class UmaClasse {  
2     int x;  
3     void f() { }  
4 }  
5  
6 class OutraClasse extends UmaClasse {  
7     void g() {  
8         f();  
9         x = 10;  
10    }  
11 }
```

Sobreposição de membros

- ▶ Uma subclasse pode sobrepor um membro da superclasse
 - ▶ Atributos
 - ▶ Métodos
- ▶ É diferente de sobrecarga
 - ▶ Por que?
- ▶ Polimorfismo

Sobreposição de membros (II)

- ▶ Uma subclasse pode ter um atributo com o mesmo nome de um atributo da superclasse
 - ▶ Até mesmo com tipos diferentes
- ▶ O atributo da superclasse pode ser acessado pela palavra `super`

```
1 class UmaClasse {  
2     int x = 1;  
3     int y = 2;  
4 }
```

Sobreposição de membros (III)

- Qual é o valor de `a` em cada atribuição?

```
1 class OutraClasse extends UmaClasse {  
2     int x = 3;  
3     void f() {  
4         int a;  
5         a = x;  
6         a = y;  
7         a = this.x;  
8         a = this.y;  
9         a = super.x;  
10        a = super.y;  
11    }  
12 }
```

Sobreposição de membros (IV)

- ▶ Ao sobrepor um método, o método da subclasse será usado em vez do método da superclasse

```
1 class UmaClasse {  
2     int f() {  
3         return 1;  
4     }  
5     int f(int x) {  
6         return x + 1;  
7     }  
8 }
```

Sobreposição de membros (V)

► Sobrecarga e sobreposição 😊

```
1 class OutraClasse extends UmaClasse {  
2     int f() {  
3         return 0;  
4     }  
5     public static void main( String args[] ) {  
6         UmaClasse a;  
7         if (args[0].equals("uma")) {  
8             a = new UmaClasse();  
9         } else {  
10            a = new OutraClasse();  
11        }  
12        System.out.println(a.f()); // Polimorfismo  
13        System.out.println(a.f(1)); // Sobrecarga  
14    }
```

Sobreposição de membros (VI)

- Pode-se utilizar o `super` para acessar um método da superclasse
 - Exclui-se a classe na hora da busca pelo método

```
1 class UmaClasse {  
2     int f() {  
3         return 1;  
4     }  
5 }  
6  
7 class OutraClasse extends UmaClasse {  
8     int f() {  
9         return super.f() + 1;  
10    }  
11 }
```

Membros Estáticos

- ▶ Para definir membros estáticos, utilize-se a palavra chave `static`
 - ▶ Um membro estático refere-se à toda a classe, e não aos objetos dessa classe.
- ▶ Atributos `static`
 - ▶ Um único elemento para todas a classe

```
1 class UmaClasse {  
2     static public int x;  
3         public int y;  
4     public void f(int v) {  
5         y = v;  
6         x = v;  
7     }  
8     ...
```

Membros Estáticos (II)

```
1  public static void main( String args[] ) {  
2      UmaClasse arr[] = new UmaClasse[2];  
3      arr[0] = new UmaClasse();  
4      arr[1] = new UmaClasse();  
5      arr[0].f(4);  
6      arr[1].f(5);  
7      System.out.println("arr[0].y_=" + arr[0].y);  
8      System.out.println("arr[1].y_=" + arr[1].y);  
9      System.out.println("arr[0].x_=" + arr[0].x);  
10     System.out.println("arr[1].x_=" + arr[1].x);  
11 }  
12 }
```

Membros Estáticos (III)

- ▶ Métodos `static` só podem acessar atributos `static`
 - ▶ A não ser que utilize um objeto

```
1 class UmaClasse {  
2     static public int x;  
3         public int y;  
4     public static void f(int v) {  
5         y = v; // NAO!  
6         x = v; // Ok!  
7     }  
8 }
```

Membros Estáticos (IV)

- ▶ Como não está no contexto de um objeto, não se pode usar o `this`
- ▶ O mesmo vale para o `super`

final

- ▶ Atributos `final` não podem ter o valor alterado
 - ▶ Ou seja, constantes!

```
1 class UmaClasse {  
2     public final int x = 20;  
3     public final int y; // Problema!!  
4     public void f() {  
5         x = 4; // NAO!!  
6     }  
7 }
```

final (II)

- ▶ Métodos `final` não podem ser sobrepostas em uma subclasse

```
1 class UmaClasse {  
2     public final void f() {  
3         // ...  
4     }  
5     public void g() {  
6         // ...  
7     }  
8 }
```

final (III)

```
1 class OutraClasse extends UmaClasse {  
2     public void g() { // Ok  
3 }  
4     public void f() { // NAO  
5 }  
6 }
```

final (IV)

- Uma classe `final` não pode ser estendida!

```
1 final class UmaClasse {  
2 }  
3  
4 class OutraClasse extends UmaClasse { // NAO  
5 }
```

abstract

- ▶ Um método pode ser declarado `abstract`
 - ▶ O corpo do método não é definido
 - ▶ A classe também deve ser declarada como `abstract`

```
1 abstract class UmaClasse {  
2     abstract public void f();  
3  
4     public int x = 10;  
5     public void g() {  
6         //  
7     }  
8 }
```

abstract (II)

- ▶ Não é possível se criar uma instância de uma classe abstrata!
 - ▶ É preciso estender a classe e sobrepor os métodos abstratos
- ▶ Uma classe que estende uma classe abstrata e que não sobrepor os métodos abstratos também é abstrata

```
1 abstract class OutraClasse extends UmaClasse {  
2     public void g() {  
3     }  
4 }  
5  
6 class MaisUmaClasse extends OutraClasse {  
7     public void f() {  
8     }  
9 }
```

abstract (III)

► Como usar uma classe abstrata

```
1 class T {  
2     void test() {  
3         UmaClasse x; // ???  
4         x = new UmaClasse(); // ???  
5         x = new OutraClasse(); // ???  
6         x = new MaisUmaClasse(); // ???  
7     }  
8 }
```

Interface

- ▶ Define um conjunto de métodos que as classes devem implementar
 - ▶ Pode ocorrer que os métodos sejam herdados
- ▶ Se algum método não é implementado
 - ▶ A classe deve ser declarada `abstract`

```
1 interface X {  
2     void f();  
3     void g();  
4 }  
5  
6 class UmaClasse implements X {  
7     public void f() { }  
8     public void g() { }  
9     public void w() { }  
10 }
```

Interface (II)

- Considere outra interface e outra classe

```
1 interface Y {  
2     void f();  
3     void h();  
4     void w();  
5 }  
6  
7 class OutraClasse extends UmaClasse implements Y {  
8     public void h() { }  
9 }
```

Interface (III)

- Quais das atribuições abaixo são válidas?

```
1  X x;  
2  Y y;  
3  UmaClasse a;  
4  OutraClasse b;  
5  z = a;  
6  a = x; // Nao  
7  a = y; // Nao  
8  x = a; // Ok  
9  x = b; // Ok  
10 x = y; // Nao  
11 y = a; // Nao  
12 y = b; // Ok  
13 y = x; // Nao
```

Anomalias de Fluxo de Dados

- ▶ Se uma variável é declarada e existe um caminho até um uso dessa variável sem passar por uma atribuição
 - ▶ O compilador acusa uma anomalia de fluxo de dados e não compila
- ▶ Leva-se em consideração os comandos `break`, `continue` e as exceções
- ▶ Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         return x + y; // Nao  
5     }  
6 }
```

Anomalias de Fluxo de Dados (II)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         if (y > 0) {  
5             x = -y;  
6         }  
7         return x + 1; // Nao  
8     }  
9 }
```

Anomalias de Fluxo de Dados (III)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         if (y > 0) {  
5             x = -y;  
6         } else {  
7             x = 1;  
8         }  
9         return x + 1; // Ok  
10    }  
11 }
```

Anomalias de Fluxo de Dados (IV)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         while (y > 0) {  
5             x = y;  
6             y--;  
7         }  
8         return x + 1; // Nao  
9     }  
10 }
```

Anomalias de Fluxo de Dados (V)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         do {  
5             x = y;  
6             y--;  
7         } while (y > 0);  
8         return x + 1; // Ok  
9     }  
10 }
```

Anomalias de Fluxo de Dados (VI)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         do {  
5             if (y > 0) {  
6                 y -= 2;  
7                 continue;  
8             }  
9             x = y;  
10            y--;  
11        } while (y > 0);  
12    return x + 1; // Nao
```

Anomalias de Fluxo de Dados (VII)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x;  
4         do {  
5             x = y;  
6             if (y > 0) {  
7                 y -= 2;  
8                 continue;  
9             }  
10            y--;  
11        } while (y > 0);  
12        return x + 1; // Ok
```

Anomalias de Fluxo de Dados (VIII)

- ▶ Código morto também é uma anomalia
- ▶ Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         return y + 1;  
4         y ++; // Nao  
5     }  
6 }
```

Anomalias de Fluxo de Dados (IX)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         if (y > 0) {  
4             return y + 1;  
5         }  
6         y++; // Ok  
7         return y;  
8     }  
9 }
```

Anomalias de Fluxo de Dados (X)

► Exemplos

```
1 class UmaClasse {  
2     int f(int y) {  
3         if (y > 0) {  
4             return y + 1;  
5         } else {  
6             return y - 1;  
7         }  
8         y++; // Nao  
9         return y;  
10    }  
11 }
```

Anomalias de Fluxo de Dados (XI)

- O compilador considera a *possibilidade* de execução, não a *executabilidade*

```
1 class UmaClasse {  
2     int f(int y) {  
3         int x = y + 1;  
4         if (x > y) {  
5             return y + 1;  
6         }  
7         y++; // Ok, mas sabemos que nao!!  
8         return y;  
9     }  
10 }
```

Tratamento de Exceções

- ▶ Permite escrever código mais robusto
- ▶ Define comandos que devem ser **tentados**
 - ▶ Define o que pode dar errado, e como tratar o erro
- ▶ Estrutura `try-catch-finally`

Tratamento de Exceções (II)

- ▶ Os problemas que podem ocorrer são chamados de exceções
 - ▶ Cada tipo de exceção é representado por uma classe
 - ▶ Derivada da classe `Exception`
 - ▶ Pode carregar informações sobre o problema que ocorreu
 - ▶ Existem muitas classes de exceções prontas
 - ▶ Pode-se criar outras específicas para a aplicação
 - ▶ Exemplos
 - ▶ `IOException`
 - ▶ `RuntimeException`
 - ▶ `FileNotFoundException`

Tratamento de Exceções (III)

- ▶ Uma exceções pode ser lançada (em inglês, *throw*)
 - ▶ Explicitamente pelo comando `throw`
 - ▶ Implicitamente por um erro de execução
 - ▶ Exceções derivadas de `RuntimeException`
- ▶ Um trecho de código que pode lançar uma exceção deve ser protegido por uma estrutura de `try`

```
1 try {  
2     FileInputStream in = FileInputStream("dados.dat");  
3 }
```

Tratamento de Exceções (IV)

- ▶ Pode-se 'catar' (do inglês, *catch*) as exceções lançadas dentro de um `try`
 - ▶ O `catch` indica a classe da exceção que ele espera
 - ▶ Se uma exceção dessa classe (ou de uma subclasse) for lançada
 - ▶ O código do `catch` é executado

```
1 InputStream in = null;  
2 try {  
3     in = new FileInputStream("dados.dat");  
4 }  
5 catch (FileNotFoundException e) {  
6     System.out.println("Arquivos_de_dados_ãno_encontrado!!");  
7     System.out.println("Usando_a_entrada_ãpadro!!");  
8     in = System.in;  
9 }
```

Tratamento de Exceções (V)

- Pode-se usar mais de um `catch` em um mesmo `try`
 - Eles serão tentados na seqüência em que aparecem

```
1  InputStream in = null;  
2  try {  
3      in = new FileInputStream("dados.dat");  
4  }  
5  catch (FileNotFoundException e) {  
6      System.out.println("Arquivos_de_dados_ãno_encontrado  
7      System.out.println("Usando_a_entrada_ãpadro!!");  
8      in = System.in;  
9  }  
10 catch (Exception e) {  
11     System.exit(1);  
12 }
```

Tratamento de Exceções (VI)

- ▶ As regras de herança devem ser observadas

- ▶ Primeiramente, as classes mais específicas

```
1  InputStream in = null;  
2  try {  
3      in = new FileInputStream("dados.dat");  
4  }  
5  catch (Exception e) {  
6      System.exit(1);  
7  }  
8  catch (FileNotFoundException e) { // Eh um tipo de Exceção  
9      System.out.println("Arquivos_de_dados_ãno_encontrado");  
10     System.out.println("Usando_a_entrada_ãpadro!!");  
11     in = System.in;  
12 }
```

Tratamento de Exceções (VII)

- ▶ Pode-se indicar um código que deve ser executado mesmo no caso de tratamento de exceções
 - ▶ Usa-se o `finally`
 - ▶ Não importa se a execução foi bem sucedida ou não

Tratamento de Exceções (VIII)

```
1 try {
2     in = new FileInputStream("dados.dat");
3 }
4 catch (Exception e) {
5     System.exit(1);
6 }
7 catch (FileNotFoundException e) { // Eh um tipo de Exception
8     System.out.println("Arquivos_de_dados_ãno_encontrado!!");
9     System.out.println("Usando_a_entrada_ãpadro!!");
10    in = System.in;
11 }
12 finally {
13     System.out.println("Sempre_executado!!");
14 }
```

Tratamento de Exceções (IX)

- ▶ Uma função pode deliberada não tratar uma exceção que pode ser levantada
 - ▶ Se a exceção for `RuntimeException`
 - ▶ Ok
 - ▶ Se não
 - ▶ Deve declarar que o seu código pode lançar exceções, e quais

```
1 import java.io.*;
2 class UmaClasse {
3     void f() { // E a çãexceo FileNotFoundException ???
4         FileInputStream in = new FileInputStream("dados.dat");
5     }
6 }
```

Tratamento de Exceções (X)

- Se não trata, deve declarar!

```
1 import java.io.*;  
2 class UmaClasse {  
3     void f() throws FileNotFoundException {  
4         FileInputStream in = new FileInputStream("dados.d  
5     }  
6 }
```

Tratamento de Exceções (XI)

- ▶ Ao usar o método `f`, sabe-se que ele pode lançar uma exceção
 - ▶ Ou trata
 - ▶ Ou declara que lança

```
1 class OutraClasse {  
2     void g() {  
3         UmaClasse a = new UmaClasse();  
4         a.f(); // E a çãexceo FileNotFoundException??  
5     }
```

Tratamento de Exceções (XII)

► Tratar

```
1 class OutraClasse {  
2     void g() {  
3         UmaClasse a = new UmaClasse();  
4         try {  
5             a.f();  
6         }  
7         catch (FileNotFoundException e) {  
8             ...  
9         }  
10    }  
11 }
```

Tratamento de Exceções (XIII)

► Declarar

```
1 class OutraClasse {  
2     void g() throws FileNotFoundException {  
3         UmaClasse a = new UmaClasse();  
4         a.f();  
5     }  
6 }
```

Tratamento de Exceções (XIV)

- ▶ Deve-se declarar a exceção mais específica possível
- ▶ Pode-se declarar mais de uma exceção

Tratamento de Exceções (XV)

- Criar uma exceção
 - Basta ser uma classe derivada de `Exception`

```
1 class NotaBaixaExcecao extends Exception {  
2     private int nota;  
3     NotaBaixaExcecao(int nota) {  
4         this.nota = nota;  
5     }  
6     int getNota() {  
7         return nota;  
8     }  
9 }
```

Tratamento de Exceções (XVI)

- ▶ Pode-se lançar uma exceção explicitamente com comando `throw`
 - ▶ Pode ser uma exceção do sistema
 - ▶ Por exemplo, *FileNotFoundException*

```
1 class UmaClasse {  
2     void f(int n) throws NotaBaixaExcecao {  
3         if (n < 3) {  
4             throw new NotaBaixaExcecao(n);  
5         }  
6         // Continua  
7     }  
8 }
```

Tratamento de Exceções (XVII)

- ▶ As mesmas regras se aplicam
 - ▶ Ou trata
 - ▶ Ou declara

```
1 class OutraClasse {  
2     void g(int n) {  
3         UmaClasse a = new UmaClasse();  
4         try {  
5             a.f(n);  
6         }  
7         catch (NotaBaixaExcecao e) {  
8             System.out.println(e.getNota());  
9         }  
10    }
```

Tratamento de Exceções (XVIII)

- ▶ As exceções de execução pode surgir a qualquer ponto
 - ▶ Tudo deveria ter um `try` ☺
 - ▶ Exemplo
 - ▶ Divisão pode 0 `ArithmeticException`
 - ▶ Acesso fora dos limites do `array` `IndexOutOfBoundsException`
- ▶ Essas exceções podem ser deixadas sem tratar
 - ▶ Mas pode-se tratá-las
 - ▶ Robustez

```
1  int arr[];  
2  ...  
3  try {  
4      arr[5] = 10;  
5  }  
6  catch (IndexOutOfBoundsException e) {  
7      System.out.println("Oops!!_Nao_existia_a_posicao_5")  
8  }
```

Tratamento de Exceções (XIX)

- ▶ Não pode ser substituto de um código bem estruturado
 - ▶ O código anterior ficaria melhor assim

```
1  int arr[];  
2  ...  
3  if (arr.length >= 6) {  
4      arr[5] = 10;  
5  } else {  
6      // Não existe a posição 5  
7  }
```

Pacotes

- ▶ Ajudam a agrupar classes relacionadas
 - ▶ Facilitam a localização e a distribuição
- ▶ As classes podem compor um pacote
 - ▶ Usa-se a declaração `package` para dizer qual é o pacote do arquivo
 - ▶ Pode-se ter mais de uma classe por arquivo, mas apenas um pacote

```
1 package PrimeiroPacote ;  
2  
3 class UmaClasse {  
4 }  
5  
6 class OutraClasse {  
7 }
```

Pacotes (II)

- ▶ Deve-se respeitar algumas regras para a organização dos fontes
 - ▶ Impostas pelos compiladores
 - ▶ Podem variar de compilador para compilador
 - ▶ Mas normalmente não variam

Pacotes (III)

- ▶ Só pode haver uma classe pública por arquivo
 - ▶ O nome de arquivo deve ser o nome da classe, seguido de `.java`
- ▶ Assim, no arquivo `UmaClasse.java`

```
1 public class UmaClasse {  
2 }  
3  
4 class OutraClasse { // Ok! Nao eh public  
5 }
```

Pacotes (IV)

- ▶ Os pacotes devem ser organizados em subdiretórios

- ▶ Por exemplo

```
1 package PrimeiroPacote ;  
2  
3 public class UmaClasse {  
4 }
```

- ▶ Deve ser colocada no arquivo `UmaClasse.java` do diretório `PrimeiroPacote`
- ▶ Não basta colocar o arquivo no diretório para ele fazer parte do pacote
 - ▶ É preciso colocar a declaração do `package` também

Pacotes (V)

- ▶ Pode-se usar um caminho para nomear os pacotes

```
1 package br.usp.icmc.labes.util;
```

- ▶ Deve ser colocado no caminho `br/usp/icmc/labes/util`

Pacotes (VI)

- ▶ Para se utilizar um pacote
 - ▶ A estrutura deve ser mantida
- ▶ Para usar uma classe

```
1 PrimeiroPacote.UmaClasse x = new PrimeiroPacote.UmaC
```

- ▶ Para importar um classe

```
1 import PrimeiroPacote.UmaClasse;  
2 UmaClasse x = new UmaClasse();
```

- ▶ Para importar todas as classes de um pacote

```
1 import PrimeiroPacote.*;
```

Pacotes (VII)

- ▶ Existem muitos pacotes padrões
 - ▶ `java.lang`
 - ▶ Importado automaticamente
 - ▶ `java.io`
 - ▶ `java.net`
- ▶ Existem muitos pacotes que pode ser obtidos (comprados, baixados)

Strings

- ▶ Uma `string` é sempre um objeto
 - ▶ Mas se pode escrever entre aspas `"hello world"`

```
1 class UmaClasse{  
2     void f() {  
3         String str1 = "hello_world";  
4         String str2 = new String(str1);  
5         char c[] = {'h','e','l','l','o','_','w','o','r','l'};  
6         String str3 = new String(c);  
7         int l1 = str1.length();  
8         int l2 = "hello".length();  
9     }  
10 }
```

Strings (II)

- ▶ A classe `Object` tem um método `toString()`
 - ▶ Retorna uma `string` que representa o objeto
 - ▶ As classes devem sobrepô-lo se necessário
 - ▶ Sempre que é necessária uma versão `string` do objeto, `toString` é chamada
 - ▶ Por exemplo
- ```
1 System.out.println(3);
```
- ▶ O 3 é convertido em "3"

## Strings (III)

- Pode-se concatenar duas `strings` com +
  - Se um operando é uma `string`, o outro será convertido com o `toString`

```
1 class UmaClasse{
2 void f() {
3 String str1 = "hello" + "_world";
4 int num = 3;
5 String str2 = "Aluno" + num + '!';
6 }
7 }
```

## Object

- ▶ `protected Object clone()`
- ▶ `boolean equals(Object obj)`
- ▶ `protected void finalize()`
- ▶ `Class getClass()`
- ▶ `int hashCode()`
- ▶ `void notify()`
- ▶ `void notifyAll()`
- ▶ `String toString()`
- ▶ `void wait()`
- ▶ `void wait(long timeout)`
- ▶ `void wait(long timeout, int nanos)`

## Wrappers

- ▶ Todas as classes derivam de `Object`
  - ▶ Portanto, pode criar uma variável do tipo `Object` e armazenar qualquer objeto
  - ▶ Porém, os tipos básicos não são objetos
    - ▶ Usa-se *wrappers* para isso

```
1 class UmaClasse {
2 void f() {
3 Object obj;
4 obj = new String("Teste");
5 obj = new java.util.Vector();
6 obj = new UmaClasse();
7 obj = 1; // ???
8 obj = new Integer(1); // Ok
9 }
10 }
```