

Programação em Java - Fundamentos

3 - Programação orientada a objetos

Citeforma

Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

jose.l.aser@sapo.pt, pedro.g.nunes@gmail.com,
paulojasm@gmail.com

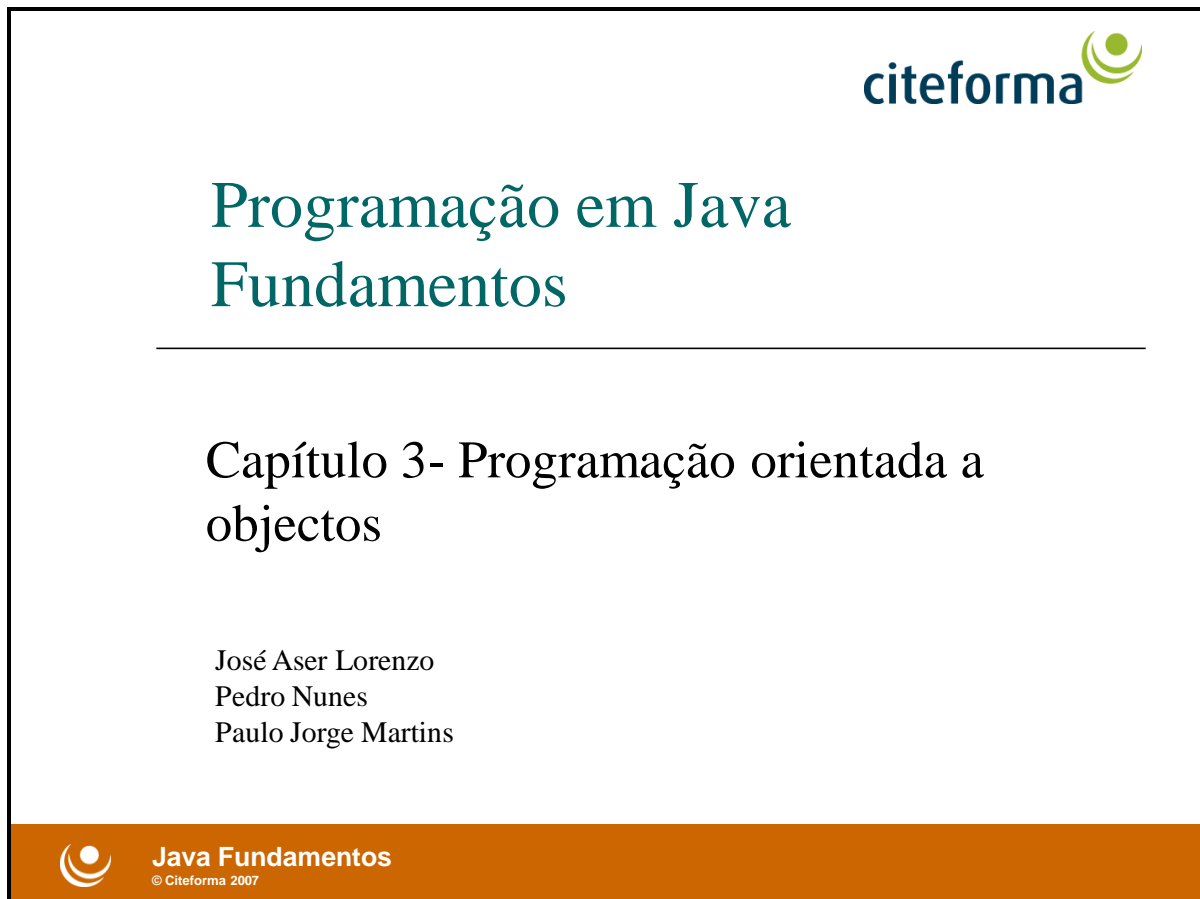
Fevereiro de 2012

Sumário

Programação orientada a objetos	4
Objetivos.....	5
Noções gerais	6
Vantagens/desvantagens da programação OO	7
Classe e Objeto.....	8
Herança	10
Classe	12
Variáveis de objeto e de classe.....	14
Métodos de objeto e de classe.....	15
Exercício sobre definição de classe: construir a classe ContaBancaria.	17
Objeto	19
Instanciar um objeto.....	20
Como um objeto acede às variáveis	21
Exercício sobre objetos	22
Exercício 1.....	22
Exercício 2.....	23
Métodos.....	24
Valor devolvido por um método	25
Método para trocar com tipos pré-definidos	26
Utilização de método troca com tipos pré-definidos.....	27
Troca trabalhando com os pointers	30
Troca com pointers trabalhando com o conteúdo	32
Exercício sobre métodos	34
this.....	36
“Overloading” de métodos.....	38
O construtor.....	40
Exercícios com construtor.....	41
Herança.....	44
O mecanismo de herança	45
Exercícios com herança	47
“Overriding” de métodos	49
Exercícios com “overriding” de métodos	51
abstract	53
Exemplo de classe abstract.....	54
Exercício com classe abstract.....	55
Exercício 1.....	55
Exercício 2.....	55
Exercício 3:.....	56
O construtor e o mecanismo de herança.....	58
O construtor por omissão	59
Exercício construtor por omissão.....	60
O construtor da super classe.....	62
“Overloading” e construtores	64
Exercício com “Overloading” e construtores	65
Exercício 1.....	65

Exercício 2.....	65
Exercício 3.....	66
Chamada entre construtores da mesma classe.....	67
Exercício: chamada entre construtores da mesma classe	68

Programação orientada a objetos



O Java está entre as linguagens de programação que seguem o paradigma da orientação a objetos.

Nos capítulos anteriores desenvolvemos programas em Java usando as suas capacidades de linguagem procedimental, reduzindo ao máximo as suas capacidades "Object Oriented". Agora que já estamos familiarizados com os aspetos básicos da sua sintaxe conhecer as suas características como linguagem orientada a objetos.

Objetivos

Objectivos

- Utilizar a linguagem Java para criar classes e instanciar objectos;
- Definir métodos que recebem parâmetros;
- Encapsular variáveis e métodos;
- Conhecer e tirar partido dos ciclos de vida das variáveis;
- Utilizar os mecanismos de herança, reescrita, e “overloading”;



No fim deste capítulo terá apreendido os conceitos de programação orientada a objetos intrínsecos à linguagem Java e ficará apto a criar classes, instanciar objetos, utilizar os mecanismos de herança, “overloading”, “overriding” (reescrita) e encapsulamento de dados.

As classes desenvolvidas nos exercícios deste capítulo deverão ficar dentro do projeto criado no capítulo anterior (**JavaFundamentos**) e dentro do package **capitulo3**.

Noções gerais

Sumário

- Noções gerais;
- Classe;
- Objecto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



Para quem não está familiarizado com uma linguagem orientada a objetos é fundamental compreender os conceitos e a terminologia utilizados por este tipo de linguagens. Primeiro veremos os conceitos de uma forma teórica geral. Depois vamos analisar cada um deles, recorrendo a exercícios para os explicar e demonstrar.

Vantagens/desvantagens da programação OO

Vantagens/desvantagens da programação OO

○ Vantagens:

- Aproximação à noção de objecto do mundo real;
- Maior modularidade no código, o que reduz a propagação de erro e a manutenção;
- Maior reutilização de código produzido;

○ Desvantagens:

- O maior nível de abstracção degrada o desempenho (será marginal?);
- Alguns programadores não gostam do paradigma OO;



O slide descreve as principais vantagens e desvantagens da programação Orientada a Objetos (OO).

Classe e Objeto

Classe e Objecto

#1/2

- Tipo de objeto:
 - Concreto - bicicleta, secretária, televisão;
 - Abstrato – uma venda, uma empresa;
- O “objeto” António possui:
 - **Estado** (variáveis): nome, idade, peso;
 - **Comportamento** (métodos): come, dorme, corre, estuda;
- A classe é uma fabrica de objetos;
- Os objetos são clones fabricados pela classe;
- Os objetos mantêm ligações à fabrica (classe);

**Java Fundamentos**
© Citeforma 2007**Capítulo 3 - Programação orientada a objects**

4

No mundo real lidamos com objetos: um cão, uma secretária, uma televisão, uma bicicleta. Estes são objetos concretos, mas no mundo real também lidamos com objetos abstratos, como uma empresa ou uma venda. Todos estes objetos possuem um estado e determinado comportamento. Por exemplo um cão possui:

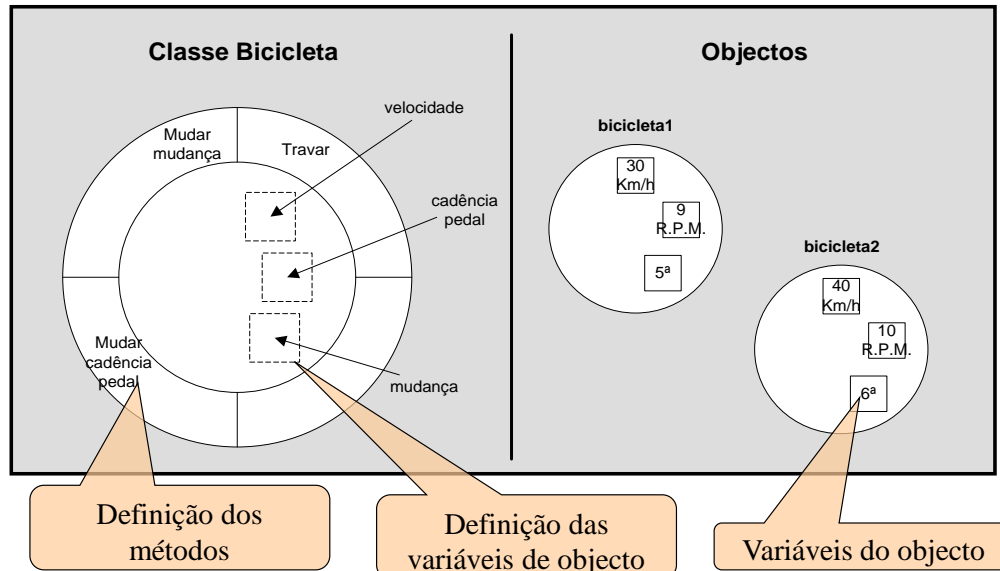
- **Estado:** nome, idade, peso;
- **Comportamento:** come, corre, dorme, ladra;

No mundo real descobrimos muito dos objetos se conhecermos as suas classes (abstrações). Por exemplo, mesmo que não saibamos o que é uma “BH”, se nos disserem que é uma marca de bicicletas, já ficamos com uma ideia geral do objeto em causa: duas rodas, pedais, travões, etc.

Os objetos criados no mundo do software são modelos do mundo real onde o estado é mantido em variáveis e o comportamento é implementado por métodos (funções). No mundo do software as classes são as fábricas de objetos. Definir a classe é definir as variáveis e os métodos que os objetos vão ter. Com as classes conseguimos produzir objetos clones uns dos outros. Mas ao contrário do mundo real, os objetos de software vão manter ligações permanentes à classe que os criou.

Classe e Objecto

#2/2



Uma bicicleta pode ser representada num programa de computador por:

- Variável velocidade instantânea, com valor 10 km/h;
- Variável cadência de pedal, com valor 90 rpm;
- Variável roda dentada atualmente escolhida para a mudança, com valor 5;
- Método travar;
- Método mudar cadência de pedal;
- Método mudar roda dentada de mudança;

Um objeto isolado não serve de muito, mas torna-se importante quando é uma “peça de uma engrenagem” na qual desempenha uma função de acordo com um plano previamente definido. Através da interação entre objetos os programadores conseguem obter as funcionalidades requeridas pela aplicação.

Os objetos de software comunicam entre si pelo envio de mensagens. Quando o objeto “a” pretende executar um dos métodos do objeto “b” procede ao envio de uma mensagem, por exemplo **b.nomeMetodo(parametro1)**. Esta é composta por três componentes:

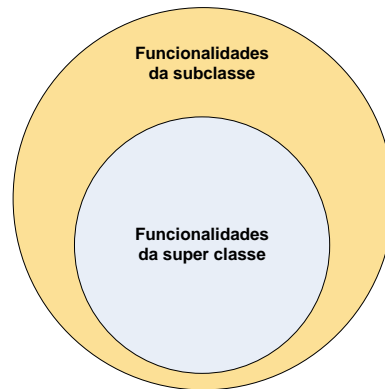
- O objeto para quem é enviada a mensagem (**b**);
- O nome do método a ser executado (**nomeMetodo()**);
- Os parâmetros requeridos pelo método (**parametro1**);

Herança

Herança

#1/2

- Permite à subclasse herdar métodos e variáveis da super classe;
- A subclasse define o que é novo ou diferente;
- A subclasse estende as funcionalidades da super classe;



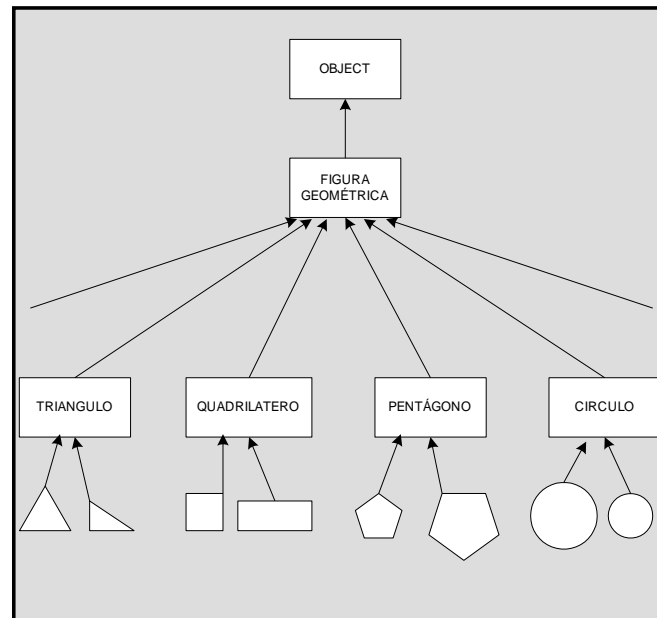
As linguagens de programação orientada a objetos implementam mecanismos de herança que permitem reaproveitar em novas classes (subclasses) as variáveis e métodos de outra classe (super classe).

A super classe é normalmente genérica, definindo um comportamento geral. Cada subclasse que vai sendo criada é especializada numa finalidade concreta. Uma subclasse herda o conjunto de variáveis e métodos da super classe. Na subclasse é definido o “comportamento” adicional em relação à super classe.

Associado à herança está a partilha de atributos e comportamentos. Se para um conjunto de classes tivermos de definir variáveis e métodos que possuem nomes e funcionalidades semelhantes, recorrer ao mecanismo de herança poderá ser uma opção para resolver o problema.

Herança

#2/2



A figura acima representa a estrutura hierárquica de um conjunto de classes. A classe **Object** ocupa o topo da hierarquia de classes na linguagem Java, e todas as classes herdam as suas características e funcionalidades.

A classe Figura Geométrica contém as características genéricas comuns a todas as figuras planas. Através do mecanismo de herança definem-se as subclasses mais concretas: Triângulo, Quadrilátero, Pentágono e Círculo. A herança implica uma extensão de funcionalidades e não a sua redução, como o prefixo sub deixa erradamente subentender.

Consideremos o método `calcularArea()` que à partida deve existir em todas as subclasses. A sua definição muda com o tipo de figura geométrica e portanto a natureza genérica da super classe não o permite definir de forma concreta. Podemos no entanto defini-lo na super classe como método **abstrato**. Isto obriga o programador à sua implementação concreta nas respetivas subclasses. As subclasses podem também redefinir os métodos herdados (**overriding**) da super classe e assim implementá-los de forma especializada.

As vantagens de utilizar os mecanismos de herança são:

- A subclasse reaproveita todo o trabalho feito na super classe;
- A subclasse adiciona um comportamento especializado;
- Podemos criar classes abstratas que impõem comportamentos nas futuras subclasses;

Classe

Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



No mundo real é frequente termos vários objetos do mesmo tipo. Por exemplo uma bicicleta é igual a muitas outras que foram feitas pelo mesmo fabricante. Utilizando a terminologia das linguagens de programação orientadas a objetos dizemos que essa bicicleta (o objeto) é uma instancia da classe (ou tipo) de objetos denominada Bicicleta.

Quando se fabricam bicicletas tira-se partido do facto delas partilharem características comuns, pelo que, a partir do molde (ou modelo) podem ser fabricadas bicicletas, todas clones uns dos outros. O molde para produzir esses objetos é a **classe**.



Uma classe é um molde ou um modelo que define as variáveis e os métodos comuns a todos os objetos desse tipo

A classe bicicleta, já usada anteriormente, define:

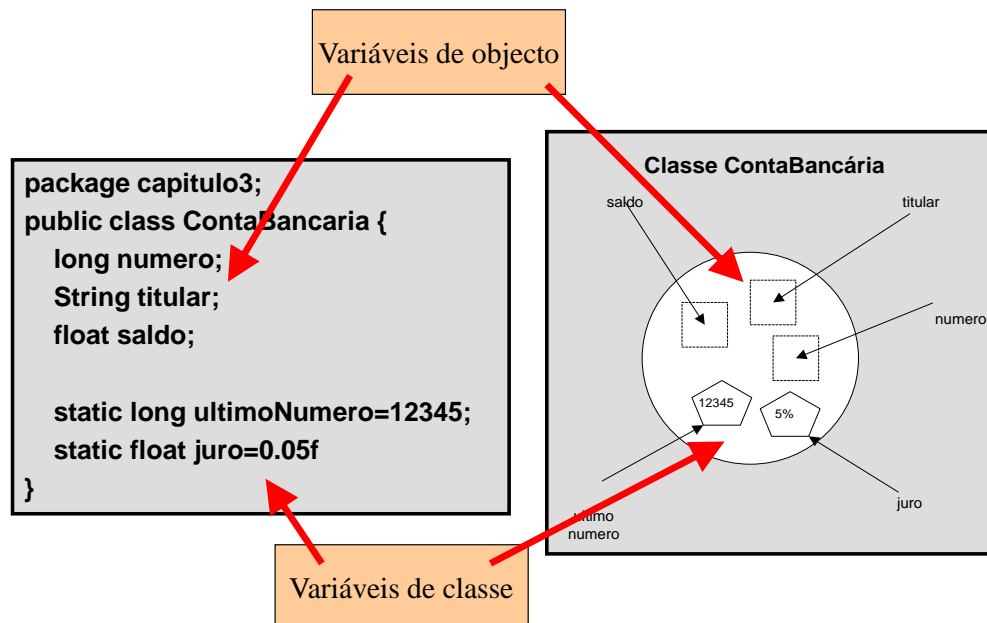
- Variáveis como: velocidade, cadência de pedal e mudança;
- Métodos como: mudar mudança, mudar cadência de pedal e travar;

Ao declarar uma classe definem-se as suas características, que são:

- Variáveis de objeto;
- Variáveis de classe;
- Métodos de objeto;
- Métodos de classe;

Variáveis de objeto e de classe

Classe: variáveis de objecto e de classe



Na figura acima encontra-se a definição da classe **ContaBancaria**:

- **Variáveis de objecto:** número de conta, titular da conta e saldo da conta;
- **Variáveis de classe:** ultimoNumero e juro.

As variáveis de objeto e de classe são declaradas dentro da classe e fora dos métodos.

As variáveis e os métodos de classe são definidos com o modificador **static**.

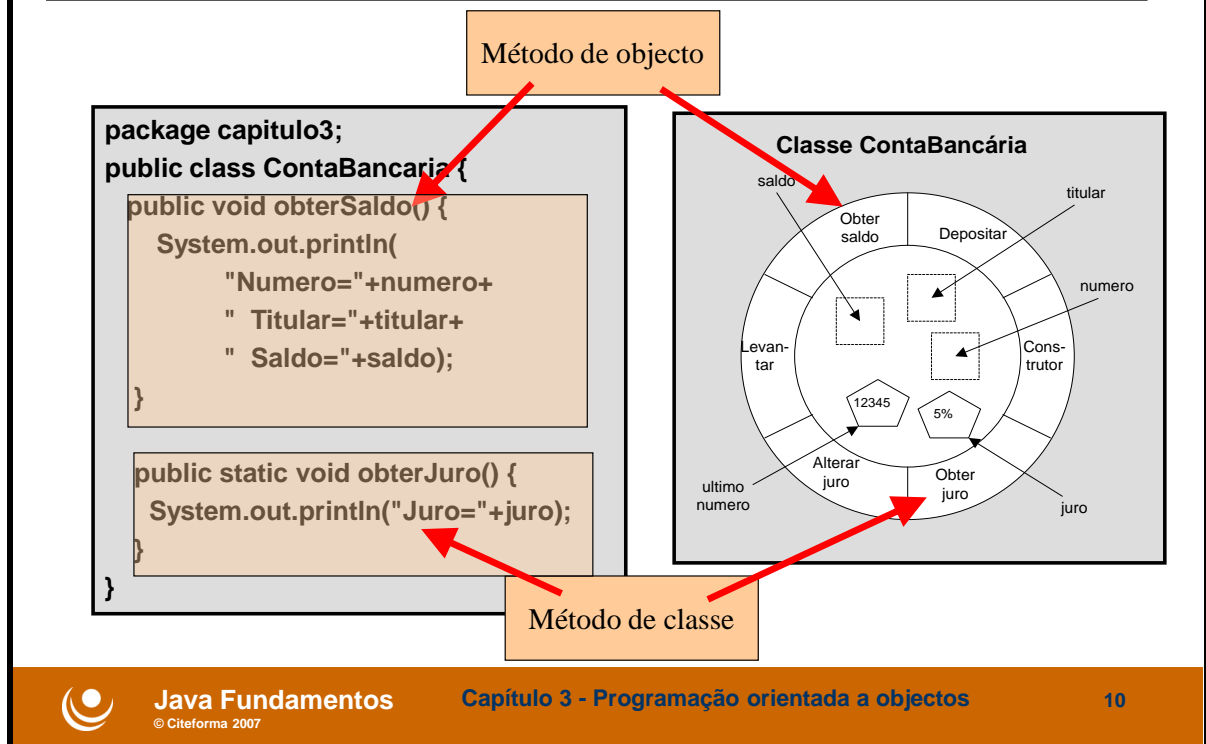
Declarar a classe implica a definição de variáveis e métodos. Cada instância dessa classe é um objeto, ao qual o computador reserva uma área de memória onde são colocadas as suas variáveis de objeto, de acordo com o que foi definido na classe.

Note que embora as variáveis do objeto sejam definidas na classe, elas só ocupam espaço depois de criados os objetos, tendo cada objeto as suas próprias variáveis.

No diagrama acima as variáveis de objeto estão representadas por um quadrado a tracejado. Usamos tracejado porque estas variáveis foram declaradas mas ainda não foram criadas, o que significa que não estão a ocupar espaço na memória. Elas apenas serão criadas quando criarmos objetos desta classe, e nessa altura, cada objeto terá as três variáveis acima definidas.

Métodos de objeto e de classe

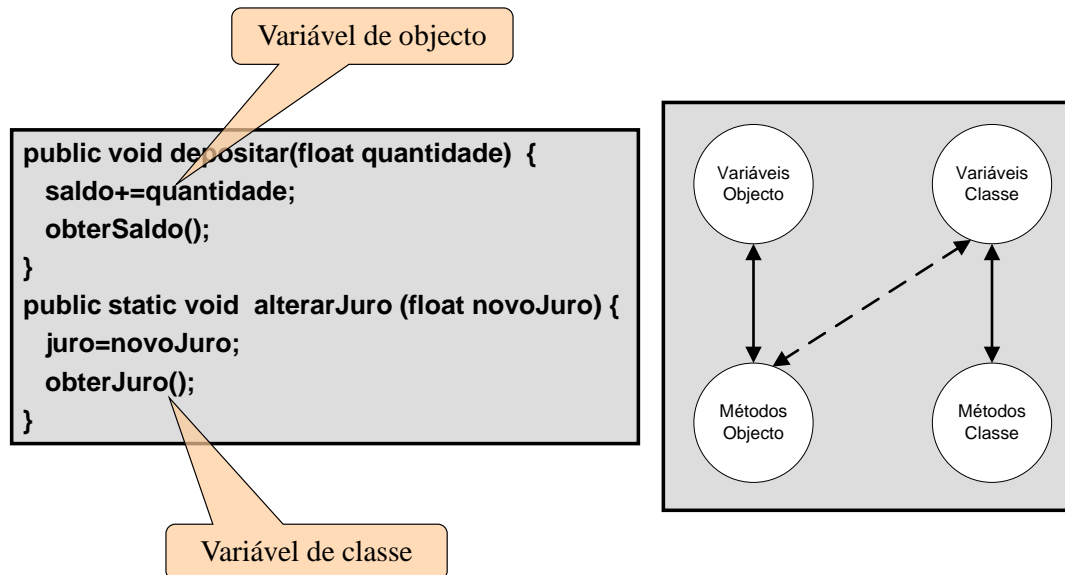
Classe: métodos de objecto e de classe



O comportamento do objeto é implementado pela execução dos métodos aplicados às variáveis desse objeto. Os métodos podem ser executados por todos os objetos dessa classe, mas manipulam apenas as variáveis do objeto que os mandou executar e no seu espaço de memória.

Os métodos de classe e as variáveis de classe são definidos com o modificador **static**, e podem ser acedidos diretamente pela classe, mesmo que não se tenha criado nenhum objeto dessa classe. Um exemplo desta invocação é `ContaBancaria.obterJuro()`

Classe: como os métodos alteram variáveis



Os métodos de objeto podem ler e alterar as variáveis de objeto e as variáveis de classe. Os métodos de classe só trabalham com as variáveis de classe. Por isto é comum afirmar-se que os métodos de objeto alteram o estado do objeto, enquanto os de classe alteram o estado da classe.

Embora os métodos de objeto consigam alterar as variáveis de classe, é uma boa prática alterar essas variáveis apenas com métodos de classe. Isto torna o código mais claro e portanto evita erros.

Um objeto A pode aceder às variáveis do objeto B. Este acesso direto pode por em perigo a integridade dos dados de B. Proteger as variáveis de um objeto pelos seus métodos é conhecido por **encapsulamento**. Esta técnica consiste em obrigar a que todas as variáveis sejam alteradas pelo recurso a métodos, e nunca por invocação direta. Esta técnica permite esconder os detalhes de implementação não importantes, conseguindo-se assim um nível de abstração como na indústria de componentes. Conseguimos também aumentar a segurança e a integridade dos dados.

Por exemplo, para alterar a mudança na bicicleta não necessito compreender como funciona o mecanismo das rodas dentadas, mas apenas saber qual a alavanca que tenho que deslocar para ativar esse mecanismo. Em objetos de software não preciso saber como os métodos foram implementados, mas sim qual o nome do método que devo chamar e quais os seus parâmetros de forma a produzir os resultados desejados nas variáveis.

Exercício sobre definição de classe: construir a classe ContaBancaria.

Exercício sobre definição de classe

- Construir a classe ContaBancaria:
 - Definir as variáveis de objeto numero, titular e saldo;
 - Definir as variáveis de classe ultimoNumero e juro;
 - Definir os métodos de objeto: obterSaldo(), depositar(quantidade) e levantar(quantidade);
 - Definir os métodos de classe: alterarJuro() e obterJuro();



Recorde que para declarar uma classe definem-se as suas características, que são:

- Variáveis de objeto;
- Variáveis de classe (com o modificador static);
- Métodos de objeto;
- Métodos de classe (com o modificador static);

A sintaxe da linguagem Java permite definir os métodos por qualquer ordem, antes ou depois das variáveis. Alguns autores recomendam que as variáveis de classe devem ser definidas antes das variáveis de objeto e antes dos métodos, para facilitar a leitura do código. Neste e nos próximos exercícios, tentaremos seguir essa recomendação.



```
package capitulo3;

public class ContaBancaria {

    /*variaveis de classe*/
    static long ultimoNumero=12345; //numero da ultima conta criada
```

```
static float juro=0.05f;           //taxa de juro para todas as contas

/*variaveis de objeto */
long numero;           //numero de conta
String titular;        //nome do titular da conta
float saldo;           //saldo de conta

//metodos para manipular as variaveis de classe
public static void alterarJuro(float novoJuro) {
    juro=novoJuro;       //ContaBancaria.juro
    obterJuro();
}

public static void obterJuro() {
    System.out.println("Juro="+juro);
}

/*métodos para manipular as variaveis de objeto */
public void obterSaldo() {
    System.out.println("Numero="+numero+" Titular="+
        titular+" Saldo="+saldo);
}
public void depositar(float quantidade) {
    saldo+=quantidade; //quantidade a depositar vem de fora
    obterSaldo();
}
public void levantar(float quantidade) {
    if (saldo>=quantidade) {
        saldo-=quantidade;
        obterSaldo();
    } else {
        System.out.print("Nao tem saldo suficiente ! ");
        obterSaldo();
    }
}

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

Objeto

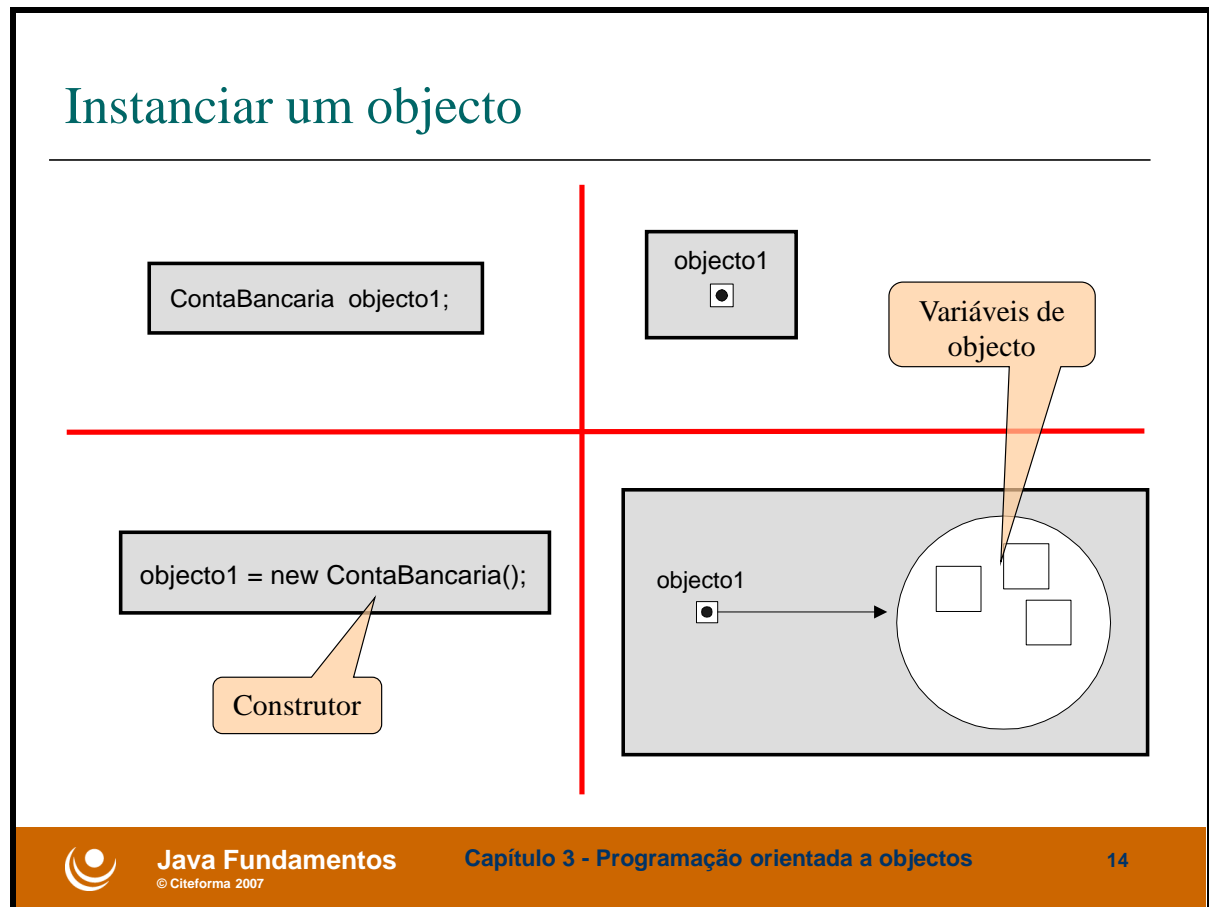
Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



Como vimos no capítulo 2 podemos criar programas sem usar objetos, mas isso impede-nos de tirar partido de todas as potencialidades da linguagem.

No exemplo anterior definimos uma classe cujo método `main()` não tinha trabalho para fazer...

Instanciar um objeto

Criar um programa em Java, significa construir um conjunto de classes que servirão como modelos para a criação de objetos. O trabalho efetuado pelo programa consiste na manipulação desses objetos.

A criação de um objeto implica:

1. A criação de uma variável que irá “apontar” para o objeto;
2. A instanciação do objeto, usando o comando **new** seguido de uma chamada a um método **construtor** da classe;

No exemplo da conta bancária estas duas instruções seriam:



```
ContaBancaria object1;  
object1 = new ContaBancaria();
```

As duas instruções podem ser condensadas numa única:



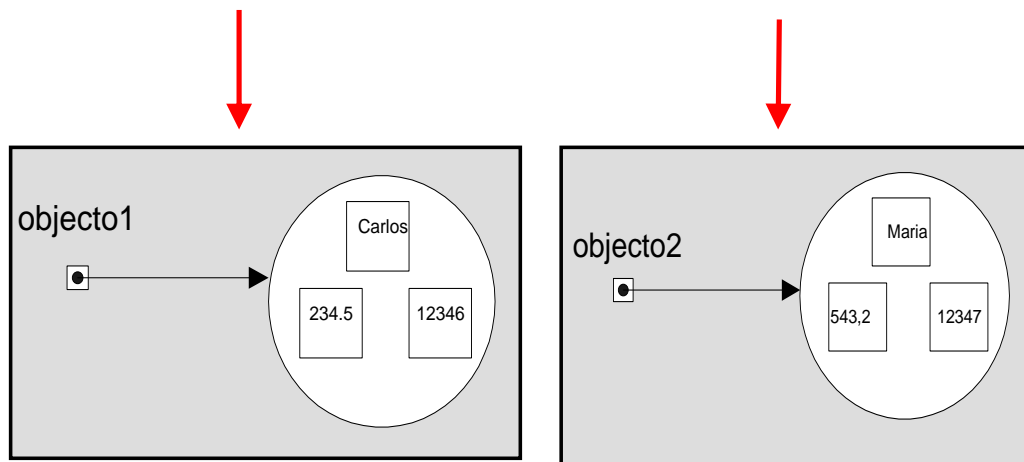
```
ContaBancaria object1 = new ContaBancaria();
```

Como um objeto acede às variáveis

Como um objecto acede às variáveis

```
objecto1.numero=++ultimoNumero;  
objecto1.titular="Carlos";  
objecto1.saldo=234.5f;
```

```
objecto2.numero=++ultimoNumero;  
objecto2.titular="Maria";  
objecto2.saldo=543.2f;
```



Um objeto tem acesso às suas variáveis e às variáveis da classe a que pertence. Para aceder às variáveis do objeto temos que referir o nome do objeto, o operador “.” e o nome da variável.

Exercício sobre objetos

Exercício sobre objectos

- Adicionar à classe ContaBancaria o método main(args) que:
 - Cria dois objetos de ContaBancaria;
 - Atribui um valor a todas as variáveis de todos os objetos;
 - Escreve o conteúdo de todas as variáveis na consola;

**Exercício 1**

Adicionar à classe ContaBancaria o método main() e dentro dele criar dois objetos de ContaBancaria.



```
package capitulo3;

public class ContaBancaria {
    //aqui fica o que já foi definido
    . . .
    . . .
    public static void main (String[] args) {
        ContaBancaria objecto1 = new ContaBancaria();
        ContaBancaria objecto2 = new ContaBancaria();
    }
}
```



No próximo exemplo são adicionadas instruções ao método main() de ContaBancaria para atribuir valores às variáveis dos objetos e consultar esses valores:

Exercício 2

Atribuir valores às variáveis dos objetos e consultar esses valores.



```
package capitulo3;

public class ContaBancaria {
    //aqui fica o que já foi definido
    . . .
    . . .
    public static void main (String [] args) {
        ContaBancaria objecto1 = new ContaBancaria();
        ContaBancaria objecto2 = new ContaBancaria();

        objecto1.numero=++ultimoNumero;
        objecto1.titular="Carlos";
        objecto1.saldo=234.5f;

        objecto2.numero=++ultimoNumero;
        objecto2.titular="Maria";
        objecto2.saldo=543.2f;

        System.out.println(objecto1.numero+" "+objecto1.titular
                           +" "+objecto1.saldo);
        System.out.println(objecto2.numero+" "+objecto2.titular
                           +" "+objecto2.saldo);
    }
}
```



Se executarmos a classe ContaBancaria obtemos o seguinte resultado:



```
12346 Carlos 234.5
12347 Maria 543.2
```

Métodos

Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



A estrutura lógica de um programa Java é definida pelas classes que ele utiliza e pelos objetos instanciados. O comportamento de cada objeto e a sua interação com os outros objetos são determinados pelos métodos.

No ponto anterior abordámos os métodos de objeto e dos métodos de classe. Os primeiros só podem ser executados havendo um objeto e podem alterar as variáveis desse objeto, o que modifica o seu estado. Estes métodos podem também alterar as variáveis da classe, embora não o devam fazer. Os métodos de classe são definidos com a palavra reservada **static**, podem ser executados sem nenhum objeto instanciado e apenas podem modificar as variáveis da classe, alterando o estado da classe.

O recurso a métodos para alterar as variáveis, tanto de objeto como de classe, dá mais controlo sobre o código produzido e permite que este seja mais fácil de alterar e menos sensível a erros. Este princípio chama-se **encapsulamento** e na prática recomenda que se defina uma interface (que utiliza métodos) para alterar o estado do objeto.



O encapsulamento pretende converter as classes em caixas negras. Isto significa que o acesso às variáveis que definem o estado do objeto é feito recorrendo a métodos definidos na interface pública da classe. O acesso direto às variáveis fica proibido, o que aumenta o controlo sobre o estado do objeto (o valor das suas variáveis)

Valor devolvido por um método

Valor devolvido por um método

- Um método é uma função;
- Uma função devolve sempre um valor;
- Esse valor é um tipo predefinido, um apontador ou void;

```
public boolean levantar(float quantidade) {  
    if (saldo >= quantidade) {  
        saldo -= quantidade;  
        return true;  
    } else {  
        return false;  
    }  
}
```

Devolve boolean

```
public void levantar(float quantidade) {  
    if (saldo >= quantidade) {  
        saldo -= quantidade;  
        obterSaldo();  
    } else {  
        System.out.print(  
            "Nao tem saldo suficiente ! ");  
        obterSaldo();  
    }  
}
```

Devolve void



Os métodos em Java são funções pelo que podem retornar um valor ou nenhum valor. Se não retornar nenhum valor coloca-se **void** na declaração do método. Se retornar um valor coloca-se o tipo do valor retornado na declaração do método e no seu código coloca-se a palavra reservada **return**, seguida do valor ou do identificador da variável que contém o valor a retornar. O termo valor refere-se aos tipos pré definidos (int, boolean, double, char, etc.) ou a objetos (variáveis do tipo apontador).

Acima mostra-se uma segunda versão do método `levantar()` que retorna **true** se o levantamento for bem sucedido e **false** caso contrário. O método `levantar()` definido anteriormente na classe `ContaBancaria` devolve void e efetua uma verificação do saldo da conta antes de levantar, invocando depois o método `obterSaldo()` para escrever o saldo no ecrã.

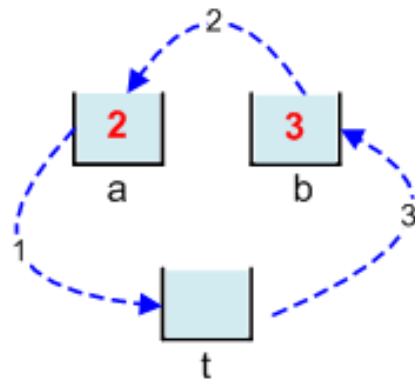
O método `obterSaldo()` é invocado no contexto do mesmo objeto que executa `levantar()`.

Método para trocar com tipos pré-definidos

Método para trocar com tipos pré-definidos

- O método abaixo pretende trocar os valores passados nos parâmetros;

```
public static void troca(int a, int b) {  
    int t;  
    System.out.println("a="+a+"    b="+b);  
    t=a;  
    a=b;  
    b=t;  
    System.out.println("a="+a+"    b="+b);  
}
```



Suponha que possui duas variáveis e que pretende trocar o seu conteúdo. No exemplo do slide definimos o método **troca** que usa tipos pré-definidos. A variável “a” vai receber o valor 2 e a variável “b” recebe o valor 3. Pretendemos trocar os conteúdos destas variáveis, de tal forma que “a” fique com 3 e “b” com 2.

O algoritmo da função faz o que se pretende, seguindo as operações esquematizadas no diagrama. Os números indicam a sequência correta, que não pode ser invertida.

Utilização de método troca com tipos pré-definidos

Utilização de método troca com tipos pré-definidos

```

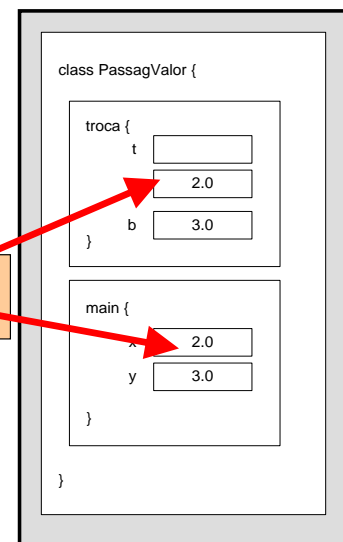
public class PassagValor {

    static void troca(float a, float b){
        float t;
        System.out.println("a="+a+" b="+b);
        t=a; a=b; b=t;
        System.out.println("a="+a+" b="+b);
    }

    public static void main(String[] args){
        float x=2.0f, y=3.0f;
        System.out.println(
            "Antes --> x="+x+" y="+y);
        troca(x,y);
        System.out.println(
            "Depois --> x="+x+" y="+y);
    }
}

```

Cópia



Quando se inicializa uma variável de um tipo predefinido (byte, short, int, long, float, double, boolean ou char), o Java guarda o seu valor na memória **Stack**. Quando se cria uma variável do tipo objeto, a sua referência (ponteiro) é também guardada na memória "Stack", mas o seu conteúdo é guardado na memória **Heap**. Os objetos do tipo String são guardados à parte numa memória chamada "**Dicionário de Strings**". Por isso, se usarmos várias vezes o mesmo conjunto de caracteres (a mesma String), na mesma classe e até no mesmo package, estamos sempre a referenciar a mesma área de memória, poupando espaço. Ver o capítulo anterior na parte referente a como comparar Strings.

Quando um método recebe parâmetros pode ter um comportamento diferente de acordo com os valores que lhe são passados. Quando passamos parâmetros para dentro de um método, o Java coloca esses valores na memória Stack, antes da chamada do método, e lê-os da Stack quando entra no método. Assim, os valores que passam para dentro de um método são sempre **cópias** de valores primitivos ou cópias de ponteiros lidos da memória Stack.

Segue o código da classe apresentada no slide:



```

public class PassagValor {

    static void troca(float a, float b){
        float t;

```

```
        System.out.println("a="+a+"    b="+b) ;  
        t=a; a=b;  b=t;  
        System.out.println("a="+a+"    b="+b) ;  
    }  
  
    public static void main(String[] args){  
        float x=2.0f, y=3.0f;  
        System.out.println("Antes  --> x="+x+" y="+y) ;  
        troca(x,y) ;  
        System.out.println("Depois --> x="+x+" y="+y) ;  
    }  
}
```

Troca com tipos pré-definidos

- Quando os parâmetros da função são do tipo pré-definido a passagem é feita por valor;
 - A função cria as suas próprias variáveis;
 - O programa principal envia valores para a função;
 - **A troca não é feita;**



O exemplo apresentado na página anterior define duas variáveis no método main(), atribui-lhes valores iniciais e usa uma função para trocar os seus valores. Embora o método troca() faça a troca dos conteúdos das duas variáveis (parâmetros) dentro da função, os conteúdos das variáveis definidas em main() não sofrem alterações, pois o método trabalha sobre uma cópia das variáveis (duas novas variáveis definidas localmente) e não sobre as variáveis definidas em main().

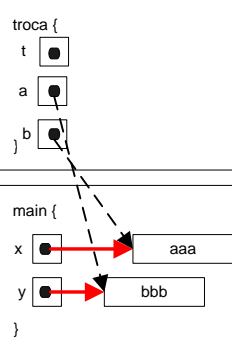
Troca trabalhando com os pointers

Troca trabalhando com os pointers #1/2



```
package capitulo3;
public class PassagValor3 {
    static void troca(String a, String b){
        String t;
        System.out.println("a="+a+"b="+b);
        t=a; a=b; b=t;
        System.out.println("a="+a+"b="+b);
    }
    public static void main(String[] args){
        String x="aaa", y="bbb";
        System.out.println("Antes-->x="+x
            +" y="+y);
        troca(x,y);
        System.out.println("Depois -->x="+x
            +" y="+y);
    }
}
```

```
class PassagRefer {
```



- “a” recebe o apontador de “x” e “b” o de “y”;
- Os valores de “a” e “b” trocam (apontadores);
- “x” e “y” continuam a apontar para o mesmo;



Segue o código apresentado no slide:

```
package capitulo3;
public class PassagValor3 {
    static void troca(String a, String b){
        String t;
        System.out.println("a="+a+"b="+b) ;
        t=a; a=b; b=t;
        System.out.println("a="+a+"b="+b) ;
    }
    public static void main(String[] args){
        String x="aaa", y="bbb";
        System.out.println("Antes-->x="+x +" y="+y) ;
        troca(x,y) ;
        System.out.println("Depois -->x="+x +" y="+y) ;
    }
}
```

Troca trabalhando com os pointers #2/2



- O parâmetro do método é uma String, logo é passado o valor do pointer;
 - A função cria um apontador;
 - O programa principal envia o valor do pointer (o endereço do objeto original);
- O método troca os pointers e não altera os conteúdos. No método chamador os pointers não mudam;
- A troca não é feita;



Neste exemplo o que passa para dentro do método troca() são cópias dos ponteiros. Após a chamada do método troca() os conteúdos das variáveis **x** e **y** não são alterados.

O diagrama do slide anterior explica porquê: quando trocamos o valor dos ponteiros **a** e **b** dentro do método troca() essa alteração é feita apenas localmente e não é refletida em main(). Não estamos a alterar o conteúdo das variáveis **x** e **y** definidas em main() mas sim os apontadores das variáveis cópia **a** e **b**.

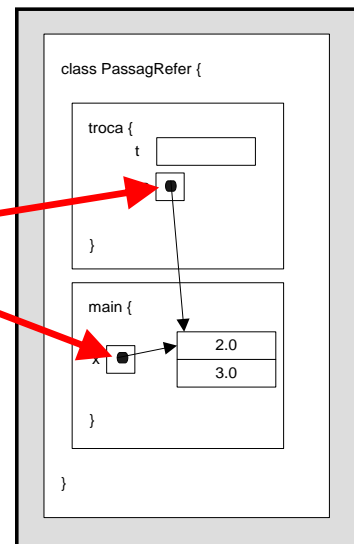
Troca com pointers trabalhando com o conteúdo

Troca com pointers trabalhando com o conteúdo #1/2 ✓

```
public class PassagValor2 {
    static void troca(float[] a){
        float t;
        System.out.println(
            "a[0]="+a[0]+" a[1]="+a[1]);
        t=a[0]; a[0]=a[1]; a[1]=t;
        System.out.println(
```

- O valor do apontador é copiado para "a";
- Ambos apontadores referenciam o mesmo objecto;
- A alteração é feita no valor do objecto;

```
float[] x= new float[2];
x[0]=2.0f;    x[1]=3.0f;
System.out.println("Antes --> x[0]="
    +x[0]+" x[1]="+x[1]);
troca(x);
System.out.println("Depois --> x[0]="
    +x[0]+" x[1]="+x[1]);
    }
}
```



Segue o código da classe apresentada no slide:

```
public class PassagValor2 {
    static void troca(float[] a){
        float t;
        System.out.println("a[0]="+a[0]+" a[1]="+a[1]);
        t=a[0]; a[0]=a[1]; a[1]=t;
        System.out.println("a[0]="+a[0]+" a[1]="+a[1]);
    }
    public static void main(String[] args){
        float[] x= new float[2];
        x[0]=2.0f;    x[1]=3.0f;
        System.out.println("Antes --> x[0]=" +x[0]+" x[1]="+x[1]);
        troca(x);
        System.out.println("Depois --> x[0]=" +x[0]+" x[1]="+x[1]);
    }
}
```


Troca com pointers trabalhando com o conteúdo #2/2 ✓

- O parâmetro do método é um array, logo é passado o valor do pointer;
 - A função cria um apontador;
 - O programa principal envia o valor do pointer (o endereço do objeto original);
- A troca é feita porque os pointers referenciam o mesmo objeto e estamos a trabalhar com os conteúdos e não com os pointers;



Neste exemplo a função recebe um pointer que referencia o objeto definido no método main(). Todas as instruções de troca lidam com o conteúdo do objeto e não com o apontador, pelo que as alterações feitas no método são refletidas no objeto do programa principal.

Exercício sobre métodos

Exercício sobre métodos

- Adicionar no método main(args) da classe ContaBancária:
 - Depositar 30,5f no objecto1;
 - Levantar 40,5f do objecto1;

 - Levantar 100,5f do objecto2;
 - Levantar 500f do objecto2;

 - Alterar o juro da classe para 0,08;
 - Invocar obterJuro();



Na classe ContaBancaria já foram definidos métodos de objeto e métodos de classe. Para os testar vamos adicionar ao método main() o que está listado acima.



```
package capitulo3;

public class ContaBancaria {
    . . .
    . . .
    public static void main (String[] args) {
        . . .
        . . .
        /*metodos de objeto */
        objecto1.depositar(30.5f);
        objecto1.levantar(40.5f);

        objecto2.levantar(100.5f);
        objecto2.levantar(500);

        /*metodos de classe*/
        alterarJuro(0.08f);
        obterJuro();
    }
}
```



A execução desta nova versão da classe ContaBancaria produz o seguinte resultado:



```
12346 Carlos 234.5
12347 Maria 543.2
Numero=12346 Titular=Carlos Saldo=265.0
Numero=12346 Titular=Carlos Saldo=224.5
Numero=12347 Titular=Maria Saldo=442.7
Nao tem saldo suficiente ! Numero=12347 Titular=Maria Saldo=442.7
Juro=0.08
Juro=0.08
```



Este exemplo mostra que a variável de classe juro pode ser alterada pelo método de classe **alterarJuro()**. O método é chamado sem nenhum objeto associado.



A utilização de métodos de objeto confirma o facto de cada objeto ter as suas próprias variáveis.



Uma análise ao método main() permite verificar que umas vezes as variáveis de objeto são alteradas diretamente (objecto1.titular="Carlos"), outras recorrendo a métodos (objecto1.depositar(30.5f)).

this**this**

- Usado nos métodos para representar o objeto atual;
- Permite distinguir o parâmetro da variável local;

```
public class This01 {  
    int x;  
    public void m1(int x) {  
        System.out.println("x="+x);  
        System.out.println("this.x="+this.x);  
    }  
    ...  
}
```

Parâmetro

Variável de
objecto

- Permite enviar o objecto actual como argumento:

```
button.addActionListener(this);
```



A palavra reservada **this** é utilizada em métodos de objeto e refere-se ao objeto em cujo contexto o método atual foi chamado.

No exemplo acima usamos a palavra reservada **this** para distinguir a variável de objeto `x` do parâmetro `x`. No contexto de um método de objeto sabemos que as operações que efetuarmos afetam o objeto ao qual esse método pertence, mas não temos nenhuma variável por nós definida que contenha o endereço desse mesmo objeto. Essa referência é dada com a palavra reservada **this**.

A palavra reservada `this` serve também para, no contexto de um método de objeto, passar o ponteiro do objeto atual como parâmetro para um método de outra classe.

Quando o operador **this** não é utilizado o Java usa a última variável que entrou na pilha de variáveis (a variável mais próxima). No exemplo acima, se indicarmos apenas `x`, estaremos a referir o parâmetro `x` e não a variável de objeto `x`.

Na classe `ContaBancaria` os métodos `levantar()` e `depositar()` chamam o método `obterSaldo()` para escrever o valor do saldo atual. Olhando para o código podem surgir dúvidas sobre qual o objeto em cujo contexto o método `obterSaldo()` será executado:



```
public void obterSaldo() {  
    System.out.println("Numero="+numero+" Titular="+
```

```
        titular+" Saldo="+saldo);  
    }  
    public void depositar(float quantidade) {  
        saldo+=quantidade; //quantidade a depositar vem de fora  
        obterSaldo();  
    }  
    public void levantar(float quantidade) {  
        if (saldo>=quantidade) {  
            saldo-=quantidade;  
            obterSaldo();  
        } else {  
            System.out.print("Nao tem saldo suficiente ! ");  
            obterSaldo();  
        }  
    }  
}
```

A palavra reservada **this** é utilizada em métodos de objeto e refere-se ao objeto em cujo contexto o método atual foi chamado. Aplicando this neste exemplo fica mais claro qual o objeto sobre o qual queremos invocar obterSaldo():



```
public void obterSaldo() {  
    System.out.println("Numero="+numero+" Titular="+  
        titular+" Saldo="+saldo);  
}  
public void depositar(float quantidade) {  
    saldo+=quantidade; //quantidade a depositar vem de fora  
    this.obterSaldo();  
}  
public void levantar(float quantidade) {  
    if (saldo>=quantidade) {  
        saldo-=quantidade;  
        this.obterSaldo();  
    } else {  
        System.out.print("Nao tem saldo suficiente ! ");  
        this.obterSaldo();  
    }  
}  
}
```

No exemplo anterior **this** torna a leitura do código mais clara, mas a sua utilização não é obrigatória. No entanto há situações em que o uso de this não é facultativo, como nos dois exemplos apresentados a seguir:

1. **x = this.x** → Quando existe uma variável de objeto e uma variável local, ambas com o mesmo nome, por exemplo x. O uso de **this** permite distinguir entre as duas, sendo this.x a variável de objeto;
2. **metodo(this)** → Permite chamar um método passando o objeto atual como parâmetro;

“Overloading” de métodos

“Overloading” de métodos

#1/2

- A assinatura de um método é:
 - O seu nome;
 - Número, tipo e ordem dos seus parâmetros;
- Em Java os métodos são identificados pela sua assinatura:
 - Podemos ter vários métodos com o mesmo nome, desde que tenham parâmetros diferentes;
 - Interessa a ordem dos parâmetros e o tipo de dados;

**Java Fundamentos**
© Citeforma 2007**Capítulo 3 - Programação orientada a objectos**

28

Overloading é o mecanismo que permite a declaração múltipla de métodos com o mesmo nome. Se considerarmos que um método é identificado pelo seu nome e pelos seus parâmetros, então podemos declarar dois métodos com o **mesmo nome** mas com **parâmetros diferentes**. Para distinguir os métodos com o mesmo nome interessa o número de parâmetros, o seu tipo e a sua ordem. Por causa deste mecanismo os métodos deixam de ser identificados apenas pelo seu nome e passam a ser identificados pela sua assinatura, sendo esta constituída por:

- Nome do método;
- Número, tipo e ordem dos seus de parâmetros;



A assinatura de um método é o que permite distinguir um método de outro. A assinatura é constituída pelo seu nome e pelo número tipo e ordem dos seus parâmetros

A palavra “*overloading*” é algumas vezes traduzida por sobrecarga. Neste manual optamos por manter a designação inglesa.

“Overloading” de métodos

#2/2

```
public class Quadrilatero {  
  
    public static void calculoArea (double lado) {  
        System.out.println("A área do quadrado é " + lado * lado);  
    }  
  
    public static void calculoArea (double lado1, double lado2) {  
        System.out.println("A área do rectangulo é " + lado1 * lado2);  
    }  
  
    public static void main (String[] args) {  
        calculoArea(3.0);  
        calculoArea(3.0,4.0);  
    }  
}
```



O exemplo do slide mostra que é possível definir dentro da mesma classe os métodos:

- calculoArea(double lado) - que calcula a área de um quadrado;
- calculoArea(double lado1, double lado2) – que calcula a área de um retângulo;



A execução do programa produz o seguinte resultado:



```
A área do quadrado é 9.0  
A área do rectangulo é 12.0
```



O “*overloading*” permite adicionar métodos que acrescentam novos comportamentos, sem alterar os que já estão definidos, o que torna as classes mais flexíveis, pois não prejudica o código que já esteja desenvolvido.

O construtor

O construtor

- É o método que permite instanciar um novo objecto da classe;
- Possui as seguintes características:
 - Tem o mesmo nome da classe;
 - Na sua definição não declaramos o tipo de valor devolvido, pois vai criar um objeto da classe;
 - É invocado com a instrução **new**;
- Se uma classe não tiver construtor, herda o construtor de **Object**;
- Alguns objetos são criados por métodos “factory”;



O **construtor** é o método que permite instanciar um novo objeto da classe. É definido dentro da classe e possui as seguintes características diferenciadoras em relação aos outros métodos:

- Tem o mesmo nome da classe (a linguagem Java é sensível a maiúsculas e minúsculas);
- Na sua definição não declaramos o tipo de valor devolvido, pois vai criar um objeto da classe;
- É invocado com a instrução **new**;

Não é obrigatório definir um construtor na classe, visto que na sua ausência a classe herda o construtor da super classe, que em última instância será o construtor da classe **Object**. Designaremos esse construtor por **construtor por omissão** que possui um comportamento muito básico: cria o novo objeto, com todas as variáveis de objeto definidas na classe e atribui-lhes os valores iniciais definidos pelo programador ou os predefinidos na linguagem.

Quando este comportamento não é suficiente o programador deve definir os seus construtores, normalmente para dar um valor inicial às variáveis de objeto diferente do predefinido e/ou para executar as instruções necessárias à manipulação inicial do objeto.

Exercícios com construtor

Exercícios com construtor

- Adicionar à classe ContaBancaria os seguintes construtores:
 - Cria uma nova conta recebendo o nome do titular e saldo;
 - Cria uma nova conta recebendo o nome do titular.
Assume que o saldo é de 50€;
 - Cria uma nova conta sem receber parâmetros. Assume que o saldo é de 50€ e coloca “Desconhecido” no nome do titular;
- Não esquecer incrementar o número de conta;



Vamos criar uma classe com o nome ContaBancaria2 que é uma cópia de ContaBancaria, mas onde colocamos 3 construtores e definimos o main() como indicado abaixo:



```
package capitulo3;

public class ContaBancaria2 {
    //variaveis de classe
    static long ultimoNumero=12345;    //numero da ultima conta criada
    static float juro=0.05f;           //taxa de juro para todas as contas
    //variaveis de objeto
    long numero;                       //numero de conta
    String titular;                    //nome do titular da conta
    float saldo;                       //saldo de conta

    //construtor de contaBancaria generico
    public ContaBancaria2(String s, float sal) {
        numero=++ultimoNumero;
        titular=s;
        saldo=sal;
    }
    //construtor de contaBancaria com nome e saldo default
    public ContaBancaria2(String s) {
        numero=++ultimoNumero;
        titular=s;
    }
}
```

```
        saldo=50f;
    }

    //construtor de contaBancaria com nome "Desconhecido" e saldo default
    public ContaBancaria2() {
        numero=++ultimoNumero;
        titular="Desconhecido";
        saldo=50f;
    }

    //metodos para manipular as variaveis de classe
    public static void alterarJuro(float novoJuro) {
        juro=novoJuro;                //ContaBancaria.juro
        obterJuro();
    }
    public static void obterJuro() {
        System.out.println("Juro="+juro);
    }

    /*métodos para manipular as variaveis de objeto */
    public void obterSaldo() {
        System.out.println("Numero="+numero+" Titular="+
            titular+" Saldo="+saldo);
    }
    public void depositar(float quantidade) {
        saldo+=quantidade; //quantidade a depositar vem de fora
        obterSaldo();
    }
    public void levantar(float quantidade) {
        if (saldo>=quantidade) {
            saldo-=quantidade;
            obterSaldo();
        } else {
            System.out.print("Nao tem saldo suficiente ! ");
            obterSaldo();
        }
    }

    public static void main(String[] args) {
        System.out.println("Construtor 1");
        ContaBancaria2 obj = new ContaBancaria2("Carlos",1.0f);
        obj.depositar(30.5f);
        obj.levantar(40.5f);

        System.out.println("Construtor 2");
        ContaBancaria2 obj2 = new ContaBancaria2("Carlos");
        obj2.depositar(0f);

        System.out.println("Construtor 3");
        ContaBancaria2 obj3 = new ContaBancaria2();
        obj3.depositar(0f);
    }
}
```



A execução deste programa produz o seguinte resultado:



```
Construtor 1
Numero=12346 Titular=Carlos Saldo=31.5
Nao tem saldo suficiente ! Numero=12346 Titular=Carlos Saldo=31.5
Construtor 2
Numero=12347 Titular=Carlos Saldo=50.0
```

```
Construtor 3  
Numero=12348  Titular=Desconhecido  Saldo=50.0
```



Comparando com o exemplo ContaBancaria executado anteriormente, neste caso foram adicionados 3 construtores e algumas instruções no método main(), que permitem criar 3 objetos usando cada um dos construtores. O “**overloading**” é muito usado nos construtores, como podemos confirmar neste exemplo, onde usamos 3 métodos diferentes para executar 3 funções diferentes, cada uma adaptada a uma situação específica.



O construtor é o único método onde não se coloca o valor retornado, nem mesmo void. Se colocar void num dos construtores acima não obtém erro de compilação, mas o método deixa de ser um construtor para passar a ser um método normal.



Um construtor pode ser **public**, **protected**, **package** ou **private**, sendo normalmente **public**. Estes qualificadores serão abordados no próximo capítulo.



Mais à frente neste capítulo falaremos do construtor e do mecanismo de herança.

Herança

Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



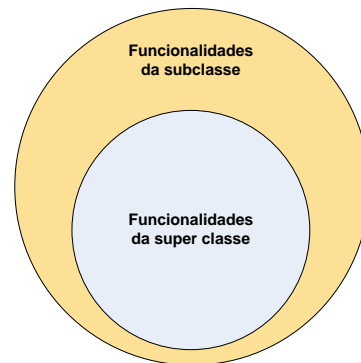
Vamos abordar a herança sobre um ponto de vista prático.

O mecanismo de herança

O mecanismo de herança

#1/2

- Permite à subclasse herdar métodos e variáveis da super classe;
- Os qualificadores dos métodos e variáveis também são herdados;
- A subclasse define o que é novo ou diferente;
- A subclasse estende as funcionalidades da super classe;

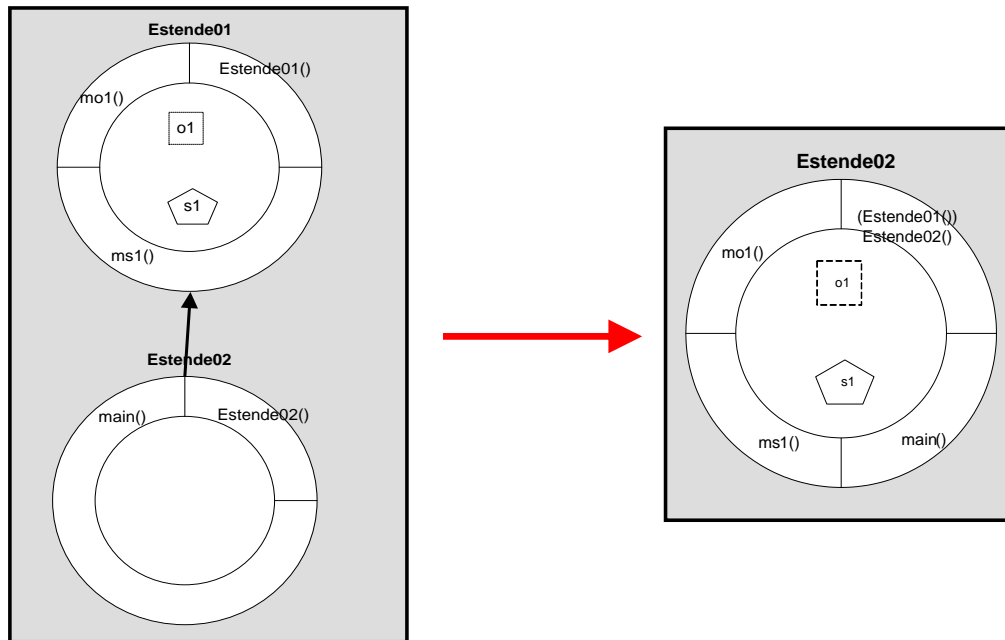


A herança é o mecanismo que permite a uma subclasse (filho):

- Herdar todas as características da super classe (pai);
- Acrescentar algumas características estendendo as da super classe (pai);
- Alterar algumas características herdadas da super classe, adaptando-se assim a funcionalidades específicas que a super classe (pai) não abrange ou não aborda de forma conveniente;

O mecanismo de herança

#2/2



O relacionamento entre as classes Estende01 e Estende02 está ilustrado no diagrama da esquerda, que mostra que a classe Estende02 herda de Estende01. Isto significa que ela possui mais características que aquelas que explicitamente define. As características funcionais de Estende02 estão ilustradas no diagrama da direita.

A herança consegue-se utilizando a palavra reservada **extends** depois do nome da classe. Em Java todas as classes são extensões da classe **Object**, umas diretamente, outras indiretamente. Quando uma classe não usa a palavra reservada **extends** torna-se uma extensão direta de **Object**. As suas subclasses serão extensões indiretas.

Exercícios com herança

Exercícios com herança

- Criar a classe **Estende01** que tem:
 - Uma variável de objecto;
 - Uma variável de classe;
 - Um método de objecto;
 - Um método de classe;
- Criar a classe **Estende02** que estende **Estende01** e adiciona o método `main()`. Criar um objeto de **Estende02** e verificar que possui as variáveis e os métodos definidos em **Estende01**;



Vamos definir a classe **Estende01**:



```
package capitulo3;


public class Estende01 {
    static int s1=3;
    int o1;

    //construtor
    public Estende01() {
        o1=4;
    }

    public static void ms1() {
        System.out.println(s1);
    }

    public void mo1() {
        System.out.println(this.o1);
    }
}
```

Em seguida vamos definir a classe Estende02 que herda de Estende01:



```
/*
 * Esta classe herda as características de Estende01. Por isso tem as
 * mesmas variáveis e métodos.
 * Esta classe adiciona o método main.
 */
package capitulo3;


public class Estende02 extends Estende01 {
    public static void main(String[] a) {
        //variáveis e métodos de classe
        System.out.println("Estende02.s1="+Estende02.s1);
        System.out.print("Estende02.ms1()="); Estende02.ms1();

        //A variável s1 é compartilhada pelas duas classes
        Estende01.s1 = 4;
        System.out.println("Estende02.s1="+Estende02.s1);

        //variáveis e métodos de objeto
        Estende02 obj = new Estende02();
        System.out.println("obj.o1="+obj.o1);
        System.out.print("obj.mo1()=");
        obj.mo1();
    }
}
```



Ao executar o método main() de Estende02 obtemos:



```
Estende02.s1=3
Estende02.ms1()=3
Estende02.s1=4
obj.o1=4
obj.mo1()=4
```



As variáveis de objeto definidas em Estende01 são herdadas em Estende02, o que significa que os objetos de Estende02 têm essas variáveis.



A variável de classe é compartilhada pelas duas classes. Uma alteração a s1 feita no contexto de Estende01 é vista na classe Estende02.

“Overriding” de métodos**“Overriding” de métodos**

#1/2

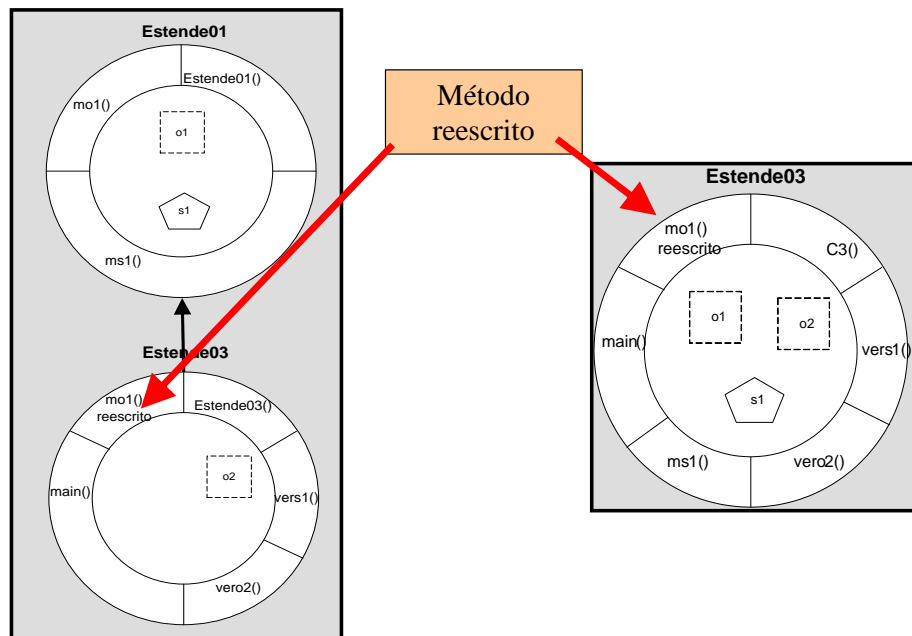
- O comportamento de um **método da super classe** pode não interessar à subclasse;
- A subclasse pode **redefinir** o método. Só tem que o reescrever;
- Os dois métodos ficam **disponíveis** aos objetos da subclasse:
 - **nome()** invoca o método redefinido;
 - **super.nome()** invoca o método da super classe;



O mecanismo de herança permite que a subclasse (filho) herde todas as características da super classe (pai). Pode acontecer que um dos métodos da super classe tenha um comportamento inadequado para a subclasse, por exemplo por ser demasiado genérico. O mecanismo de “**overriding**” permite-nos proceder à sua redefinição, ou reescrita, alterando o comportamento herdado da super classe.

“Overriding” de métodos

#2/2



No exemplo acima criamos a subclasse **Estende03** onde fazemos “overriding” do método **mo1()** herdado da super classe.

Na subclasse passamos a ter um novo comportamento, mas na realidade não estamos a descartar o método original, pois continuamos a poder chama-lo (desde que não seja abstract), fazendo **super.mo1()** em qualquer método de objeto da classe **Estende03**.

Exercícios com “overriding” de métodos

Exercícios com “overriding” de métodos

- Criar a classe **Estende03** que:
 - Adiciona o método `verS1()`;
 - Adiciona a variável de objecto `o2`;
 - Adiciona um método que permite ver a variável `o2`;
 - Redefine o método `mo1()` (escreve uma String diferente);
 - Adiciona o método `main()` que cria um objecto dela própria para testar os métodos acima;
 - Testar a invocação do método `mo1()` da super classe;



Definir a classe `Estende03`, que estende `Estende01` e redefine o método `mo1()`:



```
/*
 * Esta classe herda as características de Estende01. Por isso tem as
 * mesmas variáveis e métodos. Não é necessário defini-los.
 * Esta classe estende a super-classe, adicionando:
 *   o método de classe para ver a variável s1 da classe Estende01;
 *   a variável de objeto o2;
 *   o método para ver a variável o2;
 *   o método main, que permite criar objetos dela própria;
 * Esta classe redefine o método mo1() da super-classe, alterando-lhe
 * a funcionalidade.
 */
package capitulo3;

public class Estende03 extends Estende01 {
    int o2=5;

    public static void verS1() {
        System.out.println("C1.s1="+Estende01.s1);
    }

    public void mo1() {
```

```
//super.mol(); //como invocar o método original
System.out.println("this.o1="+this.o1);
}

public void verO2() {
    System.out.println("this.o2="+this.o2);
}

public static void main(String[] a) { //variaveis e metodos de classe
    System.out.println("Estende03.s1="+Estende03.s1);
    System.out.print("Estende03.ms1()="); Estende03.ms1();
    verS1();

    //variaveis e metodos de objeto
    Estende03 obj=new Estende03();
    obj.mol();
    obj.verO2();
}
}
```



O resultado produzido pela execução de Estende03 é o seguinte:



```
Estende03.s1=3
Estende03.ms1()=3
C1.s1=3
this.o1=4
this.o2=5
```

abstract**abstract**

- **abstract** pode ser usado em classes e métodos;
- Não é possível **instanciar objetos** de uma classe abstract;
- Quando um método é abstract **toda a classe** é abstract;
- A definição de um método abstract consiste na **definição da sua assinatura**, não tendo instruções;
- As suas **subclasses são obrigadas** a definir as instruções do método abstract. Se não o fizerem terão que ser abstract;



Quando uma classe é definida como **abstract** não é possível instanciar objetos a partir dela. O qualificador **abstract** pode ser usado em classes e métodos.

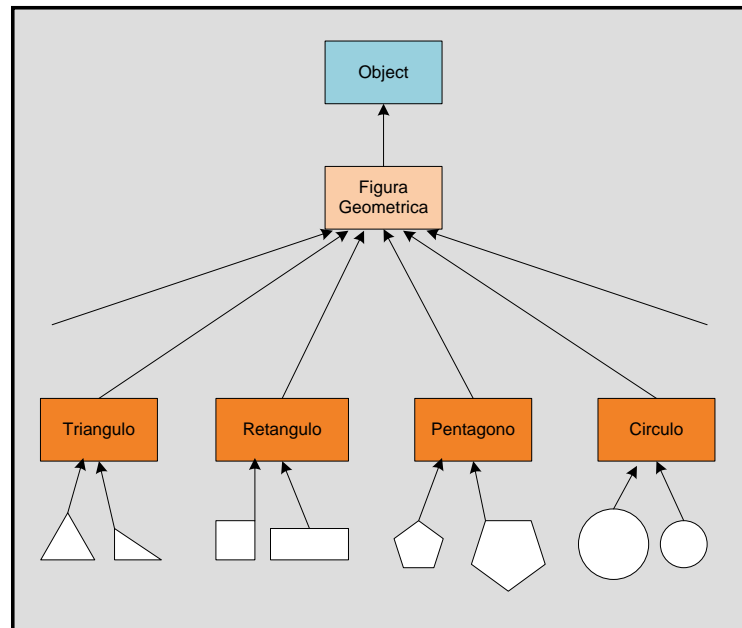
Este mecanismo é usado quando uma classe reúne características comuns a um conjunto de outras classes, tornando-a assim demasiado genérica e portanto não faz sentido instanciar objetos dela própria. Um exemplo é a classe Veículo, que tem como subclasses Ligeiro, Autocarro e Moto. Por ser demasiado genérico não se criam objetos de Veículo, mas criam-se objetos de Ligeiro, Autocarro e Moto.

Quando um método é declarado como **abstract** o programador faz uma definição do método muito simplificada: apenas tem que declarar a sua assinatura (nome e parâmetros), ficando dispensado de especificar a sua implementação. O exemplo mais à frente clarifica esta necessidade.

A existência de um método abstract obriga a classe que o contem a ser definida como abstract. As subclasses dessa classe, se não quiserem ser abstract também, terão que redefinir esse método, implementando-o. Isto cria um mecanismo de condicionamento das subclasses, pois obriga-as a implementar os métodos abstract para serem instanciadas.

Exemplo de classe abstract

Exemplo de classe abstract



Em Java todas as classes são extensões da classe **Object**, umas diretamente, outras indiretamente. Quando uma classe não usa a palavra reservada **extends**, ela torna-se uma extensão direta de **Object**. As suas subclasses serão também extensões indiretas da classe **Object**.

Vamos implementar o exemplo das figuras geométricas referido na introdução deste capítulo. Como super classe teremos a classe genérica **FiguraGeometrica**, com uma variável de objeto **área** e um método **calculaDeArea()** definido como **abstract**.

Tendo pelo menos um método abstract, a classe **FiguraGeometrica** terá de ser definida como **abstract**, não sendo por isso possível instanciar objetos dessa classe.

As subclasses **Triangulo**, **Retangulo**, **Pentagono** e **Circulo** herdam as definições de **FiguraGeometrica** e para não serem abstract terão de implementar o método **calculaArea()**.

Exercício com classe abstract

Exercício com classe abstract

- Definir a classe **FiguraGeométrica** com variável *area* e método abstract `calculoArea()`;
- Definir a classe **Retangulo** que estende **FiguraGeometrica**, possui duas variáveis **lado**, um **construtor** e o método **calculoArea()**. Adicionar método **main()** para instanciar objetos;
- Definir a classe **Circulo** que estende **FiguraGeometrica** e executa as mesmas funcionalidades que **Retangulo**;

**Exercício 1**

```
package capitulo3;

public abstract class FiguraGeometrica {
    double area;

    public abstract void calculoArea();
}
```

Exercício 2

```
/*
 * Ilustra heranca e redefinicao de um metodo abstract
 * no pai - calculoArea()
 */
```

```
package capitulo3;

public class Rectangulo extends FiguraGeometrica {
    double lado1;
    double lado2;

    //construtor
    public Rectangulo(double d1, double d2) {
        this.lado1=d1;
        this.lado2=d2;
    }

    //redefinicao de metodo herdado
    public void calculoArea() {
        this.area=lado1*lado2;
    }

    public static void main(String[] args) {
        Rectangulo r=new Rectangulo(3,4);
        r.calculoArea();
        System.out.println("Area de r="+r.area);
    }
}
```



A sua execução produz o seguinte resultado:



Area de r=12.0

Exercício 3:



```
/*
 * Ilustra heranca e redefinicao de um metodo abstract
 * no pai - calculoArea()
 */
package capitulo3;

public class Circulo extends FiguraGeometrica {
    double raio;

    //construtor
    public Circulo(double r) {
        this.raio=r;
    }

    //redefinicao de metodo herdado
    public void calculoArea() {
        this.area=Math.PI*this.raio*this.raio;
    }

    public static void main(String[] args) {
        Circulo c=new Circulo(2);
        c.calculoArea();
        System.out.println("Area de c="+c.area);
    }
}
```




A sua execução produz o seguinte resultado:



Area de `c=12.566370614359172`

O construtor e o mecanismo de herança

Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;



Sendo o construtor um método especial torna-se importante o estudo do seu comportamento perante o mecanismo de herança.

O construtor por omissão

O construtor por omissão

- Quando uma classe não tem construtor **herda o de Object**;
- Quando uma classe define um construtor **deixa de herdar** o de Object;
- O construtor da super classe é sempre invocado (**super()**);



Quando numa classe não existe a indicação expressa de **extends** ela será uma extensão de **Object**, “mãe” de todas as classes na hierarquia Java.

Uma classe que estenda **Object** e que não defina um construtor herda automaticamente o construtor definido em **Object**. Este tem um comportamento muito simples: não recebe parâmetros, instancia um objeto da classe, cria todas as variáveis de objeto que estão definidas na classe e atribui-lhes o valor predefinido na linguagem Java.

A partir do momento em que o programador define explicitamente um construtor, essa classe deixa de herdar o construtor de **Object** e o programador terá que usar o seu novo construtor.

Exercício construtor por omissão

Exercício construtor por omissão

- Definir a classe A sem construtor e a classe Main com método main() que cria objeto de A;
- Adicionar na classe A um construtor que recebe parâmetro int. Classe Main deixa de correr!!!
- Definir classe B que estende classe A. Não é possível por causa do construtor !!!
- Adicionar à classe A um construtor sem parâmetros. Tudo o que falhou começa a funcionar 😊



Classe A sem definição de construtor:



```
package capitulo3;

public class A {
    public int n=1;
}
```

Embora A não tenha um construtor definido é possível criar objetos de A em Main:



```
package capitulo3;

public class Main {
    public static void main(String[] args) {
        A o1=new A();
        System.out.println("o1.n="+o1.n);
    }
}
```



A execução de Main produz o seguinte resultado:



o1.n=1

Em seguida vamos adicionar um construtor a A, neste caso com um parâmetro int:



```
package capitulo3

public class A {
    public int n=1;

    public A (int x) {
        n=x;
    }
}
```

Ao compilar a classe Main iremos obter o erro abaixo indicado, que prova que o construtor por omissão (sem parâmetros) deixa de estar disponível:



```
Main.java:5: No constructor matching A() found in class A.    A o1=new A();
^
```

Em seguida vamos criar a classe B, que estende a classe A, e que na compilação vai esbarrar no mesmo erro:



```
package capitulo3;

public class B extends A {
    public int y=1;
}
```



```
B.java:1: No constructor matching A() found in class A.
public class B extends A
^
```

Para corrigir os dois erros de compilação obtidos anteriormente temos que adicionar um construtor sem parâmetros (construtor por omissão) à classe A:



```
package capitulo3;

public class A {
    public int n = 1;

    public A (int x) {
        this.n = x;
    }
    public A () {
    }
}
```

O construtor da super classe

O construtor da super classe

- O construtor da super classe é sempre invocado pelo construtor da subclasse;
- Isto corresponde ao método `super()`;
- Exercício:
 - Definir a classe `ConstrutorSuperClasse01` que tem um construtor que escreve texto na consola;
 - Definir a classe `ConstrutorSuperClasse02` que herda da anterior, tem um método `main()` e instancia um objeto;
 - Ao executar o `main()` de 02 verificar que o construtor de 01 é invocado;



O construtor de uma classe chama sempre o construtor da super classe antes de executar as suas próprias instruções. Para o demonstrar vamos criar as duas classes abaixo:



```
package capitulo3;

public class ConstrutorSuperClasse01 {
    public ConstrutorSuperClasse01() {
        System.out.println("Construtor de ConstrutorSuperClasse01");
    }
}
```



```
package capitulo3;

public class ConstrutorSuperClasse02 extends ConstrutorSuperClasse01{
    public ConstrutorSuperClasse02() {
        //super();
        System.out.println("Construtor de ConstrutorSuperClasse02");
    }
}
```

```
public static void main(String [] a) {  
    ConstrutorSuperClasse02 o = new ConstrutorSuperClasse02 ();  
}  
}
```



A execução de ConstrutorSuperClasse02 produz o seguinte resultado:



```
Construtor de ConstrutorSuperClasse01  
Construtor de ConstrutorSuperClasse02
```

Repare como o construtor de C2 invoca o construtor de C1 automaticamente. Isto significa que a instrução `super()` é automaticamente colocada como primeira instrução do construtor (ver o comentário).



A instrução `super()` permite invocar o construtor da super classe. A instrução `super.nomeMetodo()` permite invocar um método da super classe

"Overloading" e construtores

"Overloading" e construtores

- O mecanismo de “overloading” é muitas vezes utilizado com construtores;
- Isto significa que são criados vários métodos construtores: possuem o nome da classe mas têm parâmetros diferentes;
- Nestas situações qual será o construtor invocado?
 - Por omissão é invocado o construtor sem parâmetros: **super()**;
 - Para invocar outro é necessário colocar parâmetros: por exemplo: **super(int)**;



O mecanismo de “*overloading*” é muitas vezes utilizado com construtores. Nestas situações, qual será o construtor invocado? Os próximos exemplos mostram que por omissão é invocado o construtor sem argumentos.

Exercício com "Overloading" e construtores

Exercício com "Overloading" e construtores

- Criar a classe `ConstrutoresOverloading01` que tem dois construtores;
- Criar a classe `ConstrutoresOverloading02` que estende a anterior e cria um objecto dela própria;
- Verificar que por omissão é chamado o construtor sem parâmetros;
- Alterar o construtor de 02 de forma a que chame o construtor com parâmetros;

**Exercício 1**

Criar a classe `ConstrutoresOverloading01` que tem dois construtores.



```
package capitulo3;

public class ConstrutoresOverloading01 {
    public ConstrutoresOverloading01() {
        System.out.println("Construtor () de 01");
    }
    public ConstrutoresOverloading01(int a) {
        System.out.println("Construtor (int a) de 01");
    }
}
```

Exercício 2

Criar a classe `ConstrutoresOverloading02` que estende a anterior e cria um objeto dela própria. Verificar que por omissão é chamado o construtor sem parâmetros



```
package capitulo3;

public class ConstrutoresOverloading02 extends ConstrutoresOverloading01 {

    public ConstrutoresOverloading02() {
        System.out.println("Construtor de 02");
    }

    public static void main(String [] a) {
        ConstrutoresOverloading02 o = new ConstrutoresOverloading02();
    }
}
```



A execução de ConstrutoresOverloading02 produz o seguinte resultado:



```
Construtor () de 01
Construtor de 02
```

Exercício 3

Alterar o construtor de forma a que chame o construtor com parâmetros. De modo a chamar o construtor da super classe com argumento **int**, vamos incluir na primeira linha do construtor ConstrutoresOverloading02, a instrução **super(3)**:



```
package capitulo3;

public class ConstrutoresOverloading02 extends ConstrutoresOverloading01 {
    public ConstrutoresOverloading02() {
        super(3);
        System.out.println("Construtor de 02");
    }

    public static void main(String [] a) {
        ConstrutoresOverloading02 o = new ConstrutoresOverloading02();
    }
}
```



Após a alteração, a execução de ConstrutoresOverloading02 produz o resultado abaixo apresentado. O que mostra que foi invocado o outro construtor:



```
Construtor (int a) de 01
Construtor de 02
```

Chamada entre construtores da mesma classe

Chamada entre construtores da mesma classe

- Se a classe tiver vários construtores, um construtor pode chamar outro:
 - Usando a instrução `this(parâmetros)`;
 - Tem que ser a primeira instrução no construtor;
- Esta técnica aumenta a segurança e a flexibilidade;
- Exercício: criar a classe `ConstrutorMesmaClasse` que define 3 construtores:
 - Completo – 2 parâmetros;
 - 1 parâmetro;
 - Por omissão – sem parâmetros;



Graças ao mecanismo de “*overloading*” é possível definir vários construtores na mesma classe. Embora pouco vulgar, pode haver necessidade de que um construtor chame outro definido na mesma classe. Isto é possível em Java com a instrução **this()**, desde que esta seja a primeira instrução dentro do construtor.

O próximo exemplo ilustra esta situação; pretendemos construir dois arrays unidimensionais (vetores): um com números par e outro com números impar. Para isto são definidos três construtores:

1. O construtor principal recebe dois números e preenche os vetores começando nesses números;
2. O construtor sem parâmetros (por omissão) assume que o utilizador pretende iniciar em 2 e 3, invocando o construtor principal com 2 e 3;
3. O terceiro construtor recebe um número, assume que o utilizador pretende começar nesse número e portanto invoca o construtor principal passando esse número nos dois parâmetros;

Exercício: chamada entre construtores da mesma classe

```

/*
 * Exemplifica a chamada a um construtor dentro de outro construtor
 */
package capitulo3;

public class ConstrutorMesmaClasse {
    static final int MAX=4;
    int [] impares,pares;
    //-----
    public ConstrutorMesmaClasse() {
        this(2,3);
    }
    //-----
    public ConstrutorMesmaClasse(int n) {
        this(n,n);
    }
    //-----
    public ConstrutorMesmaClasse(int par,int impar) {
        if (par%2!=0) {
            par++;
        }
        if (impar%2==0) {
            impar++;
        }
        pares=new int[MAX];
        impares=new int[MAX];

        for (int i=0; i<MAX; i++) {
            pares[i]=par; par+=2;
            impares[i]=impar; impar+=2;
        }
    }
    //-----
    public void ver() {
        for (int i=0; i<pares.length; i++) {
            System.out.println("pares["+i+"]="+pares[i]+"\\t impares["+i+"]="+
                impares[i]);
        }
        System.out.println("-----");
    }
    //-----
    public static void main(String[] a) {
        ConstrutorMesmaClasse o1=new ConstrutorMesmaClasse(20,30);
        o1.ver();
        ConstrutorMesmaClasse o2=new ConstrutorMesmaClasse();
        o2.ver();
        ConstrutorMesmaClasse o3=new ConstrutorMesmaClasse(10);
        o3.ver();
    }
}

```

Ao utilizar **this(int, int)** estamos a referir-nos, sem nenhuma ambiguidade, ao construtor que recebe dois números inteiros como parâmetros. Esta técnica permitiu definir 3 comportamentos diferentes baseados no mesmo método, que por acaso é um construtor.



A execução de ConstrutorMesmaClasse produz o resultado apresentado a seguir:



```
pares[0]=20      impares[0]=31
pares[1]=22      impares[1]=33
pares[2]=24      impares[2]=35
pares[3]=26      impares[3]=37
-----
pares[0]=2       impares[0]=3
pares[1]=4       impares[1]=5
pares[2]=6       impares[2]=7
pares[3]=8       impares[3]=9
-----
pares[0]=10      impares[0]=11
pares[1]=12      impares[1]=13
pares[2]=14      impares[2]=15
pares[3]=16      impares[3]=17
-----
```

Sumário

- Noções gerais;
- Classe;
- Objeto;
- Métodos
- Herança;
- O construtor e o mecanismo de herança;

