Pacotes

Horácio Fernandes
horacio@icomp.ufam.edu.br

Slides: http://tinyurl.com/slides-tp

Pacotes
Introdução

- Pacotes servem para agrupar um conjunto de classes relacionadas e, possivelmente, cooperantes
  - Servem como um nível de organização
  - Programas passam a ser um conjunto de pacotes, que podem conter
    - *Classes*
    - *Interfaces*
    - *Outros pacotes*

Pacotes
Criando Pacotes

- O pacote de uma classe é definido pela palavra-chave `package`

```
package geometrico;

class Circulo {
  // Código da classe
}
```

  - Como a classe `Circulo` acima pertence ao pacote `geometrico`, o seu arquivo (`Circulo.java`) precisa estar dentro de um diretório chamado `geometrico`
  - Portanto, para “criar um pacote”, basta identificar as classes que fazem parte do pacote (usando a palavra `package` no início do arquivo) e colocar tais classes dentro de um diretório com o nome do pacote

Pacotes
Estrutura Hierárquica

- A estrutura dos pacotes é hierárquica
  - Um pacote pode conter não só classes, mas também outros pacotes
    - *que serão subpacotes do pacote atual*
    - *que serão diretórios dentro de diretórios*
  - Subpacotes podem conter outras classes e, também outros pacotes
    - *e assim por diante ...*
  - Exemplos:
    - `java.lang`
    - `java.util`
    - `java.io`
    - `br.edu.ufam.icomp`
    - `br.edu.ufam.icomp.sisca`
    - `br.edu.ufam.icomp.sisca.debug`
    - `org.openjdk.tools.compiler`
    - `gov.whitehouse.socks.mousefinder`

Pacotes
Importando Classes

- Uma classe dentro de um pacote pode usar todas as outras classes dentro do mesmo pacote
- Para usar classes de outros pacotes (incluindo subpacotes do pacote atual) é necessário importá-las:

```
// Importa a classe Circulo dentro do pacote geometrico
import geometrico.Circulo;

// Importa a classe File dentro pacote io dentro do pacote java
// Chamaremos simplesmente de pacote java.io
import java.io.File;

// Importa todas as classes do pacote java.util (LinkedList, etc)
import java.util.*;
```

- O pacote `java.lang` é automaticamente importado
  - Contém classes importantes como `String`, `System`, etc

Pacotes
Nomes de Pacotes

- Nomes de pacotes possuem todas as letras minúsculas
- Tipos de Nomes:
  - Para programas pequenos, um pacote pode não possuir nome
    - *exemplos mostrados até o momento*
    - *nestes casos, as classes devem ficar dentro do mesmo diretório, cujo nome não importa*
    - *apesar de não ter nome, o java considera que todas as classes dentro do mesmo diretório farão parte da mesma classe (que não tem nome)*
    - *este é o motivo pela qual os exemplos até o momento funcionaram sem precisar fazer um "import" das outras classes*
  - Para programas pequenos ou médios, pacotes podem ter nomes simples
    - *Exemplo:* `geometrico`, `util`, *etc*
  - Para programas maiores, que serão distribuídos mundialmente, é importante que o nome dos pacotes sejam únicos
    - *Próximo slide ...*

Pacotes
Nomes de Pacotes Únicos

- No caso de programas que serão distribuídos globalmente, desenvolvedores precisam evitar nomes de pacotes que possam vir a estar duplicados com nomes em outros programas em Java
- Convenção usada em Java para esses casos:
  - Utilizar a URL da empresa (invertido)
  - Por exemplo: IComp
    - *URL do IComp:* `icomp.ufam.edu.br`
    - *Nome de pacote único:* `br.edu.ufam.icomp`
    - *Desta forma, todos os programas em Java feitos pelo IComp estariam dentro do pacote* `br.edu.ufam.icomp`*, que poderia ter um pacote para o sistema atualmente em desenvolvimento, que poderia ter outros pacotes internos*


`br.edu.ufam.icomp`


`br.edu.ufam.icomp.sisca`


`br.edu.ufam.icomp.sisca.debug`


`br.edu.ufam.icomp.sisca.util`

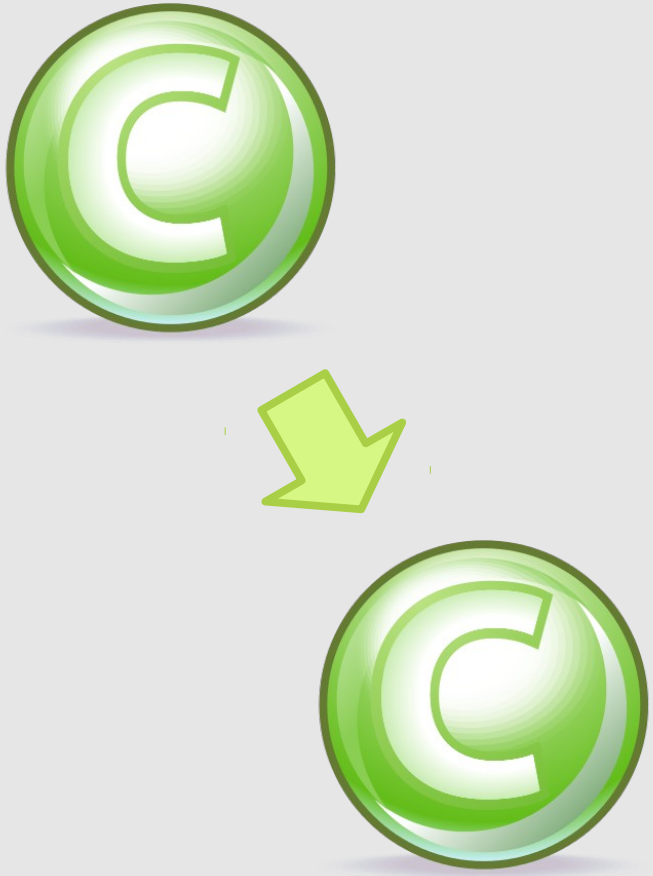

`br.edu.ufam.icomp.sisca.bd`

Pacotes
Explicando o `System.out.println`

- Agora temos todas as informações para entender

```
System.out.println("Técnicas de Programação");
```

- Classe definida no pacote `java.lang` (não precisa ser importado)
- Atributo de classe (estático) da classe `PrintStream`
- Método de instância da classe `PrintStream`
- String literal (objeto) da classe `String` (alocada no reservatório de strings)



Herança

ICOMP
Instituto de Computação

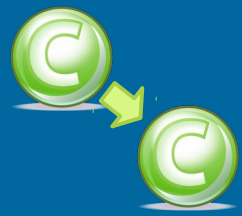
Horácio Fernandes
horacio@icomp.ufam.edu.br

Slides: http://tinyurl.com/slides-tp

Herança
Introdução

- Na Herança, a implementação de uma classe é derivada a partir de uma outra, conhecida como superclasse (ou classe pai)
  - A nova classe é conhecida como subclasse (ou classe filha)
- Herança permite que uma classe seja descrita a partir de outra já existente
  - Tanto os atributos quanto os métodos implementados na superclasse passam a fazer parte da subclasse
- A subclasse passa a ser uma espécia de subtipo da superclasse

Herança
Vantagens



- Herança é um mecanismo poderoso em linguagens orientadas a objetos
- Principais vantagens:
  - Reutilização de códigos
    - *Todo o código implementado na classe pai pode ser reutilizado na classe filha como se o código estivesse na própria classe*
    - *O compartilhamento de recursos leva a ferramentas melhores e produtos mais lucrativos*
  - Organização
    - *Classes passam a ter uma hierarquia*
  - Alterar o comportamento de uma classe
    - *É possível criar uma classe filha que é igual a classe pai, mas com um comportamento diferenciado (métodos sobrescritos)*
    - *Sem precisar mudar o código da classe original*

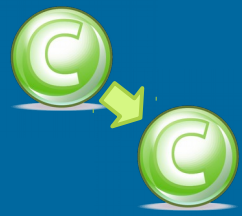
Herança
Possibilidades

- O que pode ser feito com Herança
  - Atributos e métodos da superclasse podem ser usados na subclasse diretamente como qualquer outro e atributos/métodos adicionais podem ser declarados na subclasse
  - Métodos da superclasse podem ser reimplementados na subclasse
    - *Isso é conhecido como sobreposição (visto adiante)*
  - Objetos das subclasses podem ser referenciados como sendo objetos de qualquer superclasse
    - *Isso é conhecido como generalização (visto adiante)*
    - *E o comportamento diferenciado dos métodos sobrepostos nas subclasses é conhecido como polimorfismo (visto adiante)*
  - Métodos muito genéricos podem ser declarados sem implementação
    - *Os métodos serão implementados nas subclasses*
    - *São conhecidos como métodos abstratos (visto adiante)*

Herança
Quando Usar

- Herança cria uma relação de “é um” entre uma superclasse e a subclasse
  - Se isso não ocorrer, o uso da herança não é válido
- Por exemplo:
  - A frase “uma camisa é uma roupa” expressa um uso *válido* de herança entre a superclasse `Roupa` e a subclasse `Camisa`
  - A frase “um chapéu é uma meia” expressa um uso *inválido* de herança entre as classes `Chapeu` e `Meia`

Herança
Exemplos



- Alguns exemplos de herança

Classe `Carro`

- *CarroLuxuoso*
- *CarroComercial*

Classe `Conta`

- *Corrente*
- *Poupança*

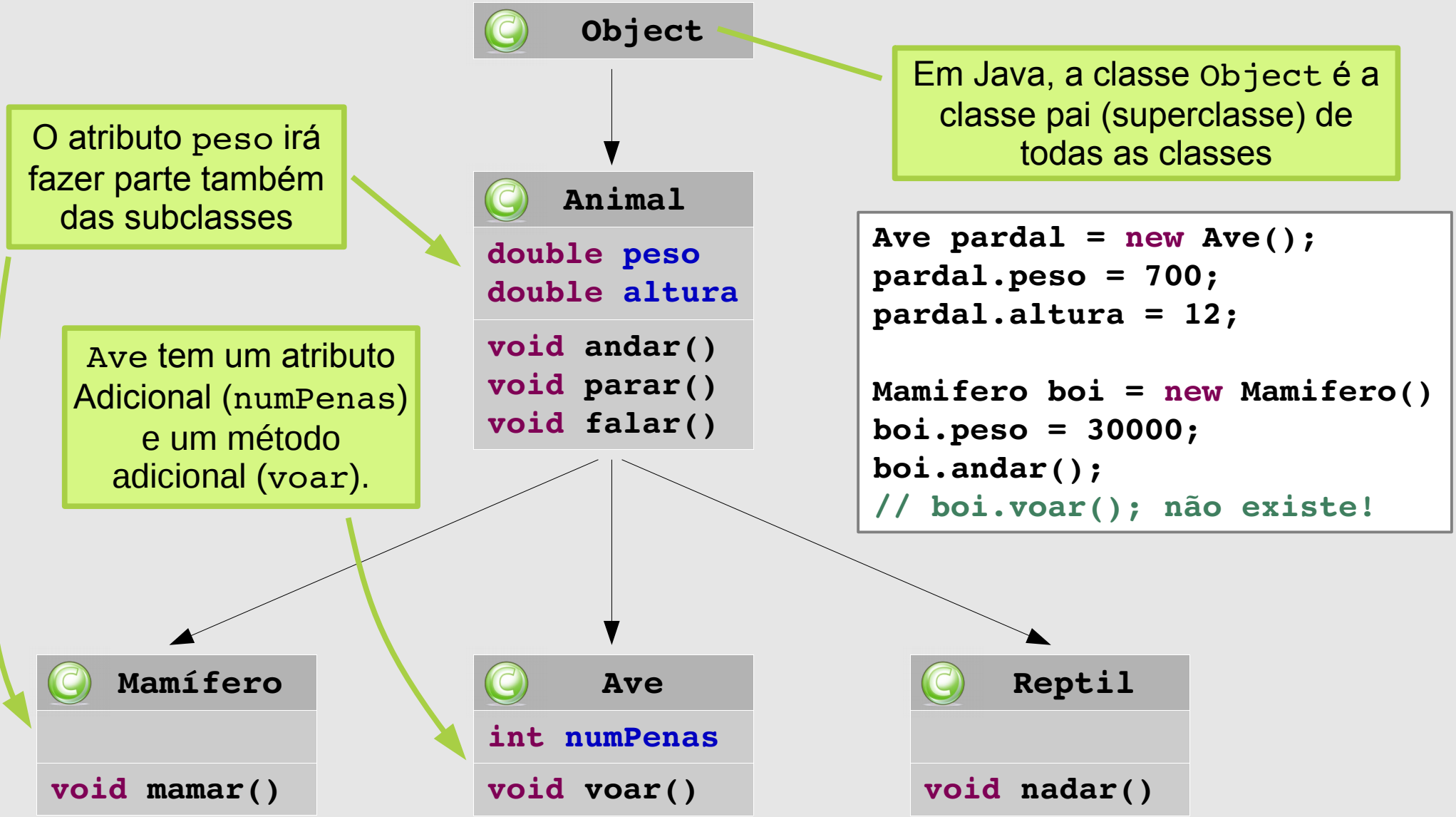
Classe `Pessoa`

- *Estudante*
  - *Graduação*
  - *Mestrado*
- *Professor*
  - *Assistente*
  - *Adjunto*
  - *Titular*
- *Funcionário*
  - *Técnico*
  - *Superior*
  - *Administrativo*

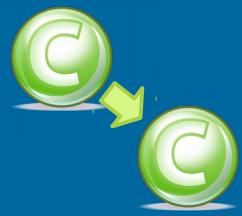
Classe `Jogo`

- *RPG*
  - *MMORPG*
- *Estratégia*
  - *TempoReal*
- *Simulação*
  - *Avião*
  - *Carro*
  - *Bode*

Herança
Exemplo



Herança
Sintaxe

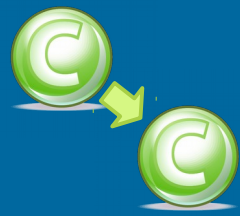


- Para especificar herança em Java, usa-se a palavra `extends`

```
public class Mamifero extends Animal {
    // Novos atributos ...
    // Novos métodos ...
}
```

  - É permitido apenas uma superclasse
    - *não há herança múltipla em Java. Motivo: Classe A tem um método "façaAlgo" e a classe B tem um método "façaAlgo" e a classe C herda A e B. O que aconteceria se alguém chamasse c.façaAlgo()?*
  - Cada classe apresenta exatamente uma superclasse
    - *exceção:* `java.lang.Object`
  - Caso não exista a cláusula `extends`, então, assume-se que a superclasse é a classe `Object`

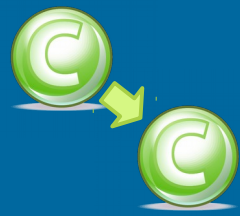
Herança
A Classe `Object`



- A classe `Object`, que faz parte do pacote `java.lang`, forma a raiz da hierarquia de classes
  - Direta ou indiretamente, toda classe é uma subclasse da `Object`
  - Define o comportamento básico que todo objeto em Java deve possuir
  - É a única classe que não possui uma superclasse
- Alguns métodos úteis:

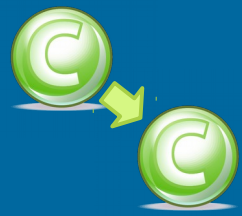
```
// Método que gera uma descrição do objeto em uma String.
// Normalmente é sobreposto (próximos slides), pois
// por padrão, imprime uma espécie de referência
String toString();

// Método que compara o objeto atual com outro. Normalmente
// é sobreposto, pois por padrão compara referências.
boolean equals(Object outro);

// Método que duplica (clona) o objeto atual
Object clone();
```

Herança
Construtores das Subclasses



- Um construtor da subclasse, caso não chame outro construtor da classe atual (usando o `this`), deve necessariamente chamar um construtor da classe pai
  - Isso deve ocorrer na primeira linha do construtor
    - *Antes do código do construtor atual*
  - Isso é feito através da chamada `super(…)`
  - Quando nenhuma chamada ao `super` é feito, o Java automaticamente inclui a chamada ao `super` sem parâmetros "`super()`" na primeira linha
    - *Isso quer dizer que se você não especificar nada, o construtor da subclasse atual irá chamar o construtor da superclasse sem parâmetros*
    - *Se o construtor sem parâmetros não existir na superclasse, ocorrerá um erro de compilação*

Herança
Construtores das Subclasses



```
public class Animal {
    double peso, altura;

    Animal(double peso, double altura) {
        this.peso = peso;
        this.altura = altura;
    }
    // Métodos andar, parar, falar ...
}
```

Como não foi especificado, o Java Irá incluir a seguinte linha aqui: `super();`
Esta linha chama o construtor da superclasse (classe `Object`)

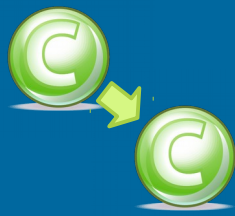
```
public class Mamifero extends Animal {
    Mamifero(double peso, double altura) {
        super(peso, altura);
    }
    // Método mamar ...
}
```

Neste caso, como o `super` foi especificado, o construtor correspondente da super classe será chamado

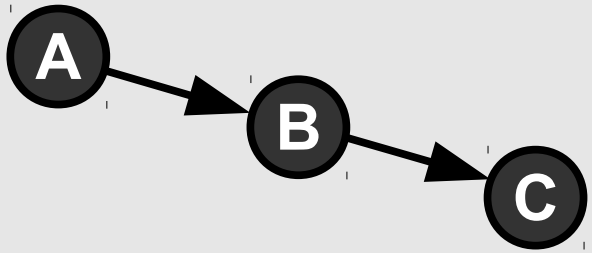
```
public class Ave extends Animal {
    int numPenas;

    Ave(double peso, double altura, int numPenas) {
        super (peso, altura);
        this.numPenas = numPenas;
    }
    // Método voar ...
}
```

Note como primeiro os atributos da superclasse são inicializados. Somente depois os atributos da subclasse serão inicializados

Herança
Construtores das Subclasses – Quiz 1



```
class A {
  int i;

  A() {
    i = 1;
  }

  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```

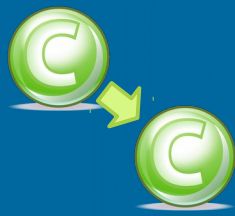



```
class B extends A {
  B() {
    i++;
  }
}
```

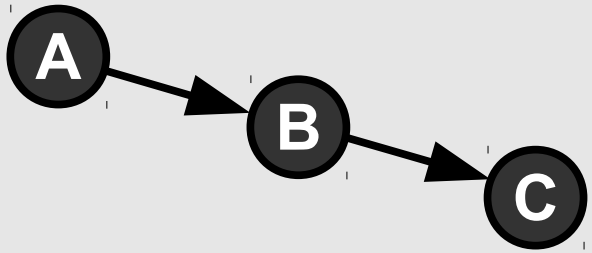
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
$ java A
a.i=1, b.i=2, c.i=3
```

Herança
Construtores das Subclasses – Quiz 2



```
class A {
  int i;

  A() {
    i = 1;
  }

  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```

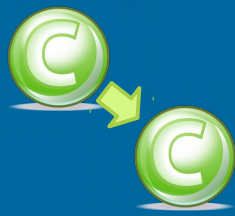



```
class B extends A {
  B() {
    i = 8;
  }
}
```

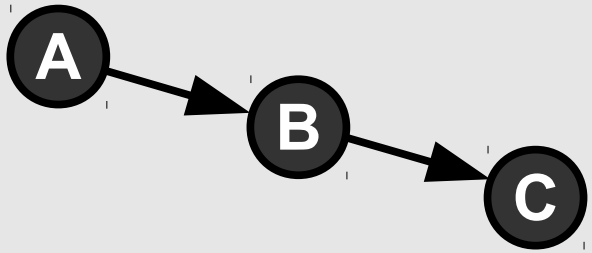
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
$ java A
a.i=1, b.i=8, c.i=9
```

Herança
Construtores das Subclasses – Quiz 3



```
class A {
  int i;

  A() {
    i = 1;
  }

  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```

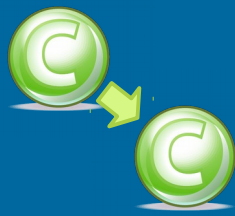



```
class B extends A {
  B() {
    i++;
  }
}
```

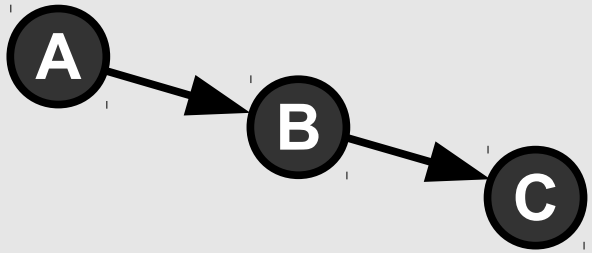
```
class C extends B {
  C() {
    i++;
    super();
  }
}
```

```
$ javac A.java B.java C.java
(...) Error (...)
Constructor call must be the first statement in a constructor (...)
```

Herança
Construtores das Subclasses – Quiz 4



```
class A {
  int i;

  A() {
    i = 1;
  }

  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```



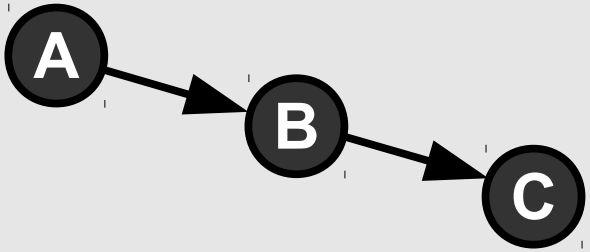

```
class B extends A {
  B() {
    i++;
  }
}
```

```
class C extends B {
  C() {
    super();
    i++;
  }
}
```

```
$ javac A.java B.java C.java
$ java A
a.i=1, b.i=2, c.i=3
```

Herança
Construtores das Subclasses – Quiz 5

```
class A {
  int i;



  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```



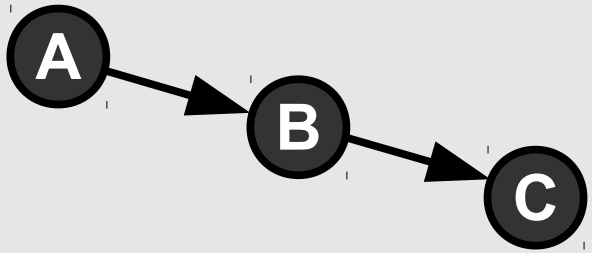

```
class B extends A {
  B() {
    i++;
  }
}
```

```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
$ java A
a.i=0, b.i=1, c.i=2
```

Herança
Construtores das Subclasses – Quiz 6

```
class A {
  int i;

  A(int a) {
    i = a;
  }

  public static void main(String[] args) {
    A a = new A(5);
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```

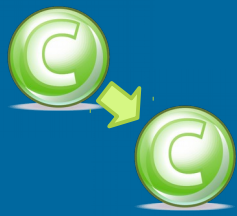



```
class B extends A {
  B() {
    i++;
  }
}
```

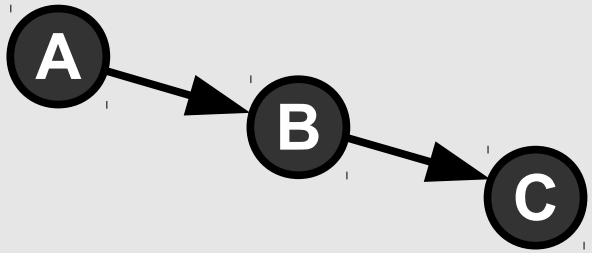
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
(...)
B.java:2: cannot find symbol
```

Herança
Construtores das Subclasses – Quiz 7



```
class A {
  int i;

  A(int a) {
    i = a;
  }

  public static void main(String[] args) {
    A a = new A(5);
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```

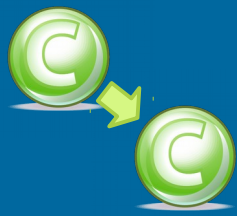



```
class B extends A {
  B() {
    super(6);
    i++;
  }
}
```

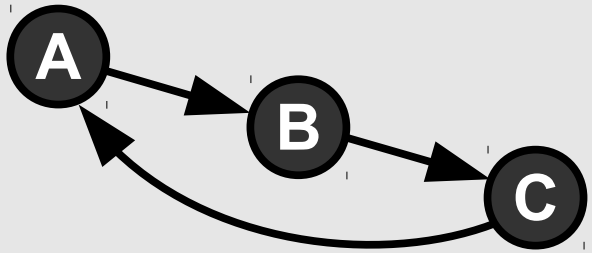
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
$ java A
a.i=5, b.i=7, c.i=8
```

Herança
Construtores das Subclasses – Quiz 8



```
class A extends C {
  int i;

  A(int a) {
    i = a;
  }

  public static void main(String[] args) {
    A a = new A(5);
    B b = new B();
    C c = new C();
    System.out.println("a.i=" + a.i
                       + ", b.i=" + b.i
                       + ", c.i=" + c.i);
  }
}
```




```
class B extends A {
  B() {
    i++;
  }
}
```

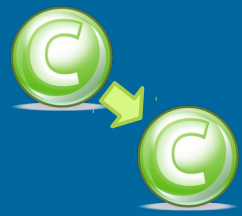
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
(...) Error (...) The hierarchy of the type A is inconsistent
Cycle detected: a cycle exists in the type hierarchy between B and A
```

Herança
Sobreposição de Métodos

- Na aula anterior, vimos que métodos poderiam ter o mesmo nome, desde que tenham parâmetros diferentes
  - Isso foi chamado de *sobrecarga*
- Na presença de herança, podemos criar métodos com o mesmo nome e com os mesmos parâmetros!
  - Entretanto, um método precisa estar na *superclasse*
  - E o outro método precisa estar na *subclasse*
  - Chamamos isso de *sobreposição*

Herança
Sobreposição de Métodos



- Na *sobreposição*, métodos possuem o mesmo nome e os mesmos parâmetros mas estão em subclasses diferentes
  - Dizemos que o método da subclasse sobrepõe o método da superclasse
  - O método da subclasse pode "reescrever" o método da superclasse
    - *Terá implementação (um comportamento) diferente*
- Exemplo
  - A classe `Object` declara e implementa o método `equals`:

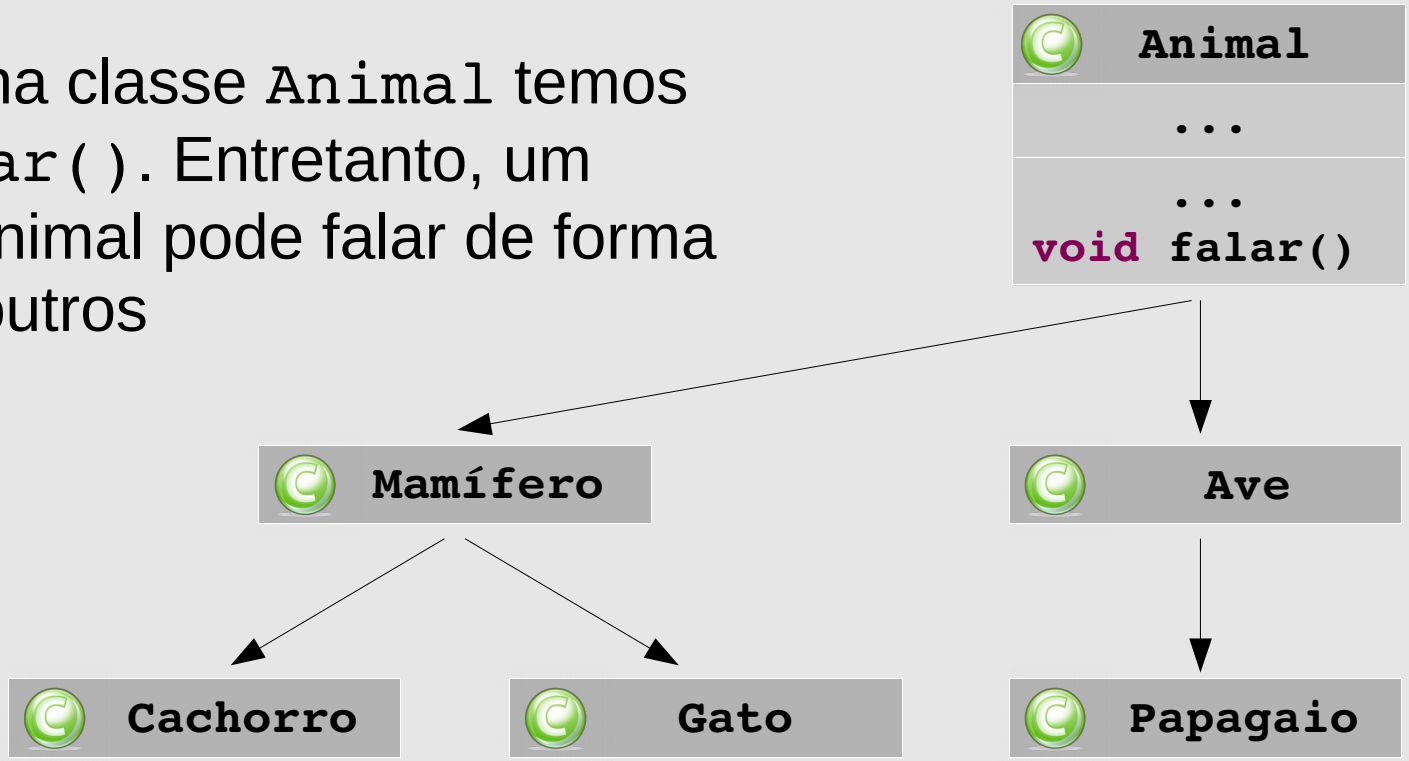
```
public boolean equals(Object obj) {
  return (this == obj);
}
```

  - Já a classe `String`, que herda a `Object` sobrepõe tal método com uma implementação diferente (compara o conteúdo da string)

```
public boolean equals(Object obj) {
  // Comparação do conteúdo (veja como no código fonte)
}
```

Herança
Sobreposição de Métodos

- A principal ideia da sobreposição é permitir que uma subclasse herde um método da superclasse e implemente-a de forma diferente
- Por exemplo, na classe `Animal` temos o método `falar()`. Entretanto, um determinado animal pode falar de forma diferente dos outros



Herança
Sobreposição de Métodos

```
class Animal {
  // Atributos, construtor padrão ...
    void falar() {
      System.out.println("Animal fala ...");
    }
  // Métodos andar, parar ...
}
```

Implementação genérica do método `falar`

```
class Cachorro extends Mamifero {
  void falar() {
    System.out.println("Cachorro fala: Auau!");
  }
}
```

`Cachorro` sobrepõe o método `falar`

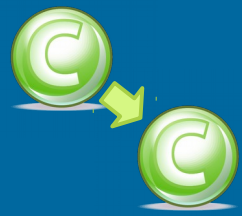
```
class Gato extends Mamifero {
  void falar() {
    System.out.println("Gato fala: Miau!");
  }
}
```

`Gato` sobrepõe o método `falar`

```
class Papagaio extends Ave {
  void falar() {
    System.out.println("Papagaio fala: Flamengo campeão!");
  }
}
```

`Papagaio` sobrepõe o método `falar`

Herança
Sobreposição de Métodos

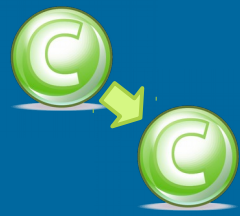


```
Animal    animal =    new Animal();
Cachorro cachorro = new Cachorro();
Gato      gato =      new Gato();
Papagaio papagaio = new Papagaio();

animal.falar();
cachorro.falar();
gato.falar();
papagaio.falar();
```

```
Animal fala ...
Cachorro fala: Auau!
Gato fala: Miau!
Papagaio fala: Flamengo campeão!
```

Note como a implementação do método `falar`, implementado inicialmente na classe `Animal` foi reescrita nas subclasses `Cachorro`, `Gato` e `Papagaio`

Herança
Generalização



- Variáveis que referenciam uma determinada classe podem apontar para qualquer objeto daquela classe *ou de qualquer subclasse dela*
- Referenciar objetos de uma subclasse como se fossem de uma superclasse (que é mais genérica), é conhecido como *generalização*

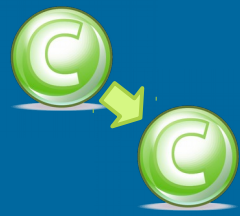
```
Animal   outroAnimal = new Cachorro();
Mamifero mamifero = cachorro;

Animal animais[] = new Animal[7];
animais[0] = cachorro;
animais[1] = gato;
animais[2] = papagaio;
animais[3] = cachorro;
animais[4] = animal;
animais[5] = outroAnimal;
animais[6] = mamifero;
```

Classe `Animal` referenciando objeto da classe `Cachorro`

Classe `Mamifero` referenciando objeto da classe `Cachorro`

Vetor de objetos da classe `Animal`. Tudo que “é um” animal (objetos das subclasses) poderá ser armazenado neste vetor

Herança
Generalização e *Casts* (Conversões)



- Quando usamos classes genéricas
  - apenas os métodos da classe genérica podem ser executados, mesmo que o objeto pertença a uma classe mais específica que tenha o método

```
Animal louroJose = new Papagaio();
louroJose.andar();
louroJose.voar();
```

OK, pois `andar` pertence a `Animal`

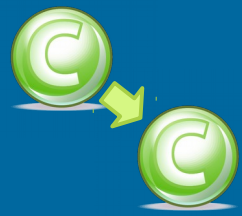
ERRO! Pois `andar` pertence à classe `Papagaio` e `louroJose` é uma referência para a classe `Animal`

- Para usar uma referência genérica como algo mais específico
  - (para podermos usar os métodos mais específicos)
  - É necessário fazer um *cast* (conversão)

```
Papagaio louroJosePapagaio = (Papagaio) louroJose;
louroJosePapagaio.voar();
```

Cast, conversão

Herança
Generalização e *Casts* (Conversões)



- Cuidado ao fazer *casts*
  - Se você tentar fazer um *cast* para uma classe que não seja a mesma do objeto ou alguma de suas superclasses, dará erro de execução

```
Cachorro louroJoseCachorro = (Cachorro) louroJose;
```

ERRO! Pois `louroJose` é um objeto da classe `Papagaio`, e não da classe `Cachorro`. E `Cachorro` não é um `Papagaio` (não é subclasse)

  - Para evitar erro de conversões, antes de fazer um *cast*, verifique se o objeto é uma instância da classe de destino

```
if (louroJose instanceof Papagaio) {
  Papagaio louroJosePapagaio = (Papagaio) louroJose;
}

if (louroJose instanceof Cachorro) {
  Cachorro louroJoseCachorro = (Cachorro) louroJose;
}
```

Verdadeiro

Falso

Herança
Generalização e Polimorfismo

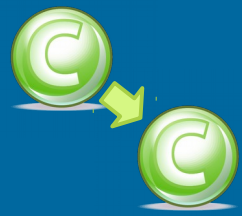
- Generalização permite tratar objetos de classes específicas (subclasses) de forma genérica (superclasse)
- Entretanto, os métodos sobrepostos dos objetos se comportarão sempre de acordo com a sua classe específica
  - Ou seja, os métodos que foram sobrepostos serão executados de acordo com a implementação da classe do objeto, e não do tipo referenciado
  - Isso é conhecido como *polimorfismo* (múltiplas formas)

Herança
Generalização e Polimorfismo

```
Animal animais[] = new Animal[7];
animais[0] = cachorro;
animais[1] = gato;
animais[2] = louroJose;
animais[3] = cachorro;
animais[4] = animal;
animais[5] = outroAnimal;
animais[6] = mamifero;

for (int i=0; i<animais.length; i++) {
  animais[i].falar();
}
```

```
Cachorro fala: Auau!
Gato fala: Miau!
Papagaio fala: Flamengo campeão!
Cachorro fala: Auau!
Animal fala ...
Cachorro fala: Auau!
Cachorro fala: Auau!
```

Herança
"Sobreposição" de Atributos



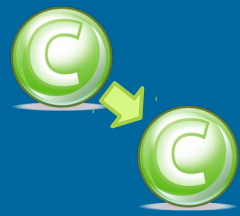
- Sobre os *atributos*:
  - Não existe sobreposição de atributos.
  - Entretanto, uma subclasse pode ter atributos com os mesmos nomes de atributos da superclasse. Dizemos que o atributo da subclasse "esconde" o atributo da superclasse, que poderá ser acessado usando o `super`

```
class A {
  int i = 1;
}
class B extends A {
  int i = 2;
  void imprimir(int i) {
    System.out.println("i = " + i + ", this.i = " + this.i
                                  + ", super.i = " + super.i);
  }
  public static void main(String args[]) {
    B b = new B();
    b.imprimir(3);
  }
}
```

Apesar de ser sintaticamente correto, esconder um atributo não é recomendado

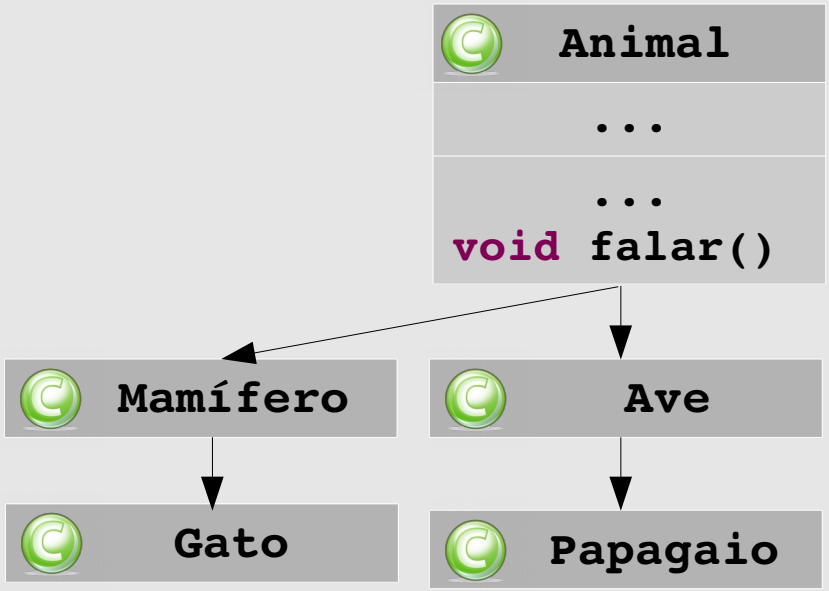
```
$ javac A.java B.java
$ java A
i = 3, this.i = 2, super.i = 1
```

Herança
Métodos Abstratos



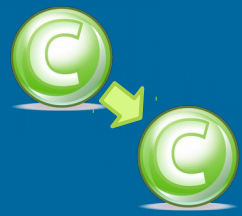
- Em uma hierarquia de classes, quanto mais alta a classe, mais abstrata é sua definição
  - Em alguns casos, alguns métodos serão tão genéricos, que fica difícil definir uma implementação útil ou que não seja específica de uma subclasse

- Por exemplo:
  - No caso da classe Animal, o método falar será implementado como?
  - No entanto, faria sentido tirar o método falar da classe? Todos os animais não falam?



- Para isso, em OO é possível definir métodos sem implementá-los!

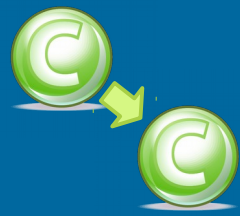
Herança
Métodos Abstratos



- Métodos Abstratos:
  - Não possuem corpo (implementação)
  - Apresentam apenas a definição seguida de “;”
  - Apresentam o modificador `abstract`
  - Se uma classe possui pelo menos um método abstrato, então ela passa a ser uma *classe abstrata* deve ser marcada como tal:

```
abstract class Animal {
    // Atributos, construtor ...

    abstract void falar();

    // Outros métodos (abstratos ou não)
}
```

Herança
Classes Abstratas



- Classes abstratas não podem ser instanciadas
  - Se criássemos um objeto da classe abstrata `Animal` e executássemos o método `falar`, como o método seria executado, se ele não possui implementação?
  - Portanto, não podemos criar um objeto de uma classe abstrata

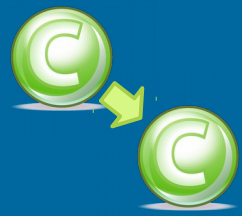
```
Animal a = new Animal();
```

```
$ javac AnimalMain.java
AnimalMain.java:3: error: Animal is abstract;
cannot be instantiated
```

- *Mas podemos usar a classe abstrata para referenciar objetos de qualquer uma de suas subclasses, como fizemos na generalização*

```
Animal a = new Cachorro();
```

Herança
Classes Abstratas



- Classes abstratas são sempre superclasses, precisam ser herdadas (estendidas) para poderem ser usadas
  - Uma subclasse de uma classe abstrata só pode ser concreta (não-abstrata) se ela sobrepor todos os métodos abstratos e fornecer implementação para cada um deles
  - Caso contrário, ela também deverá ser abstrata

```
abstract class Mamifero extends Animal {
  // Construtor e método mamar ...
}
```

A classe `Mamifero` não implementa o método `falar`, portanto deve ser abstrata

```
class Cachorro extends Mamifero {
  // Construtor ...
  void falar() {
    System.out.println("Auau!");
  }
}
```

A classe `Cachorro` implementa o método `falar`, portanto não precisa ser abstrata e pode ser instanciada

Pacotes e Herança
Laboratório 4

- http://tinyurl.com/slides-tp
  - → Laboratórios
  - → TP – 4oLaboratorio.pdf

- Entrega por E-Mail
  - Para: horacio.fernandes@gmail.com
  - Cópia: moyses.lima@icomp.ufam.edu.br
  - Assunto: TP: 4o Lab

- Data Limite:
  - Hoje, às 12hs
  - E-Mails recebidos após 12hs não serão considerados