Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação

Pacotes

Horácio Fernandes
horacio@icomp.ufam.edu.br

ColabWeb: bit.ly/pp-colabweb

Pacotes

- 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*

Criando Pacotes

- Em Java, um pacote corresponde a um **diretório**
  - Os arquivos da classe precisam estar dentro do diretório
- 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 modificar as classes que fazem parte do pacote, colocando a palavra `package` no início do arquivo, e salvar tais classes dentro de um diretório com o nome do pacote

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.beans
br.edu.ufam.icomp.beans.labs
org.openjdk.tools.compiler
```

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.*;
```

java.lang

- O pacote `java.lang` é **automaticamente importado**
  - Contém classes importantes, algumas que você já está usando:

```
String
Boolean
Byte
Double
Float
Integer
Long
Short
```

```
System
Math
Exception
IndexOutOfBoundsException
NullPointerException
Object
Comparable
Thread
```

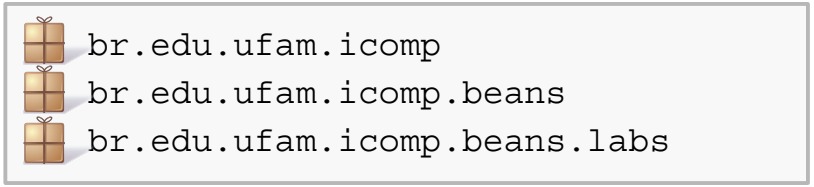
  - Para todas as outras classes em outros pacotes, há a necessidade de se usar o `import`

Nomes de Pacotes

- Nomes de pacotes possuem todas as letras minúsculas
- 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 do mesmo pacote (que não tem nome)**
  - este é o motivo pela qual os seus programas até o momento funcionaram sem precisar fazer um “import” das suas outras classes
- Para programas **médios**, pacotes podem ter nomes simples
  - Exemplo: `geometrico`, `util`, etc

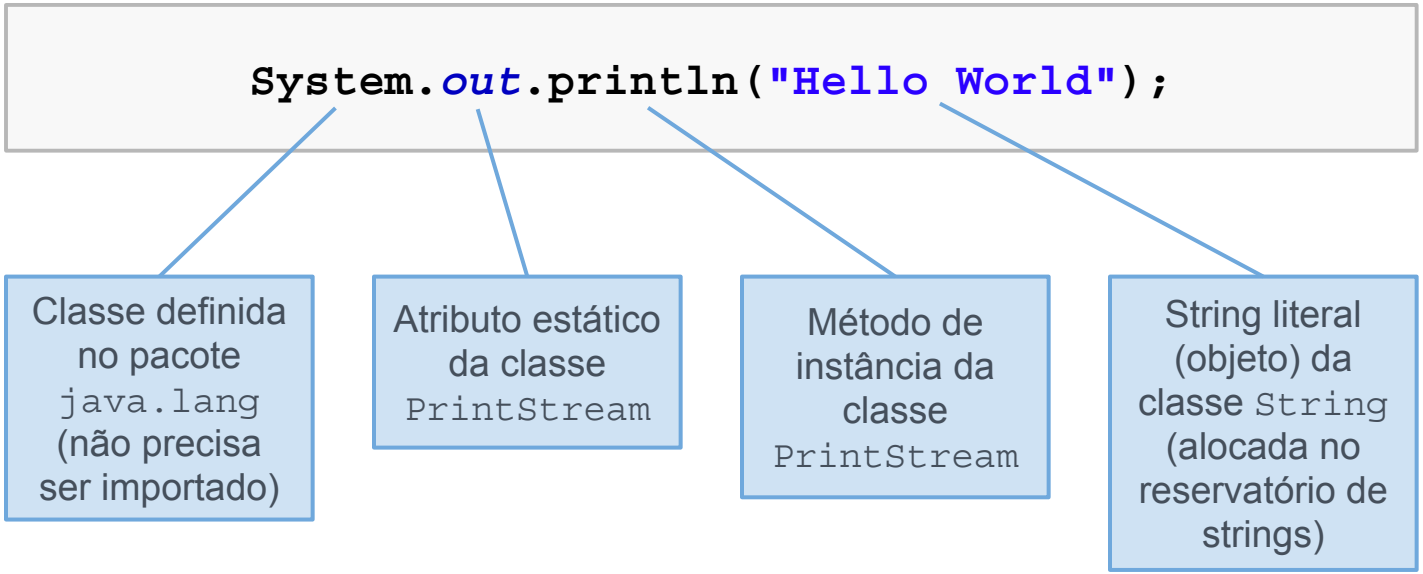
Nomes de Pacotes

- Para **programas maiores**, que serão distribuídos mundialmente, é importante que o nome dos pacotes sejam **únicos**
- 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 e que poderia ter outros pacotes internos

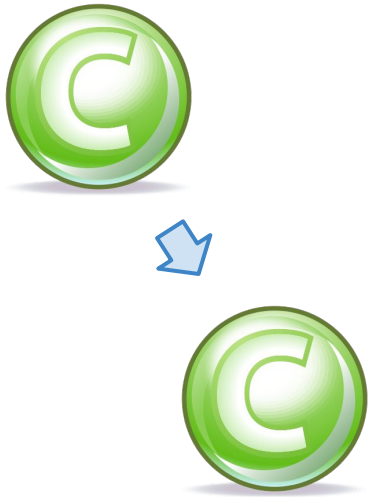


System.out.println

- Agora temos todas as informações para entender



Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Herança

Horácio Fernandes
horacio@icomp.ufam.edu.br

ColabWeb: bit.ly/pp-colabweb

Herança

- 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

Herança

- A subclasse passa a ser uma espécie de **subtipo** da superclasse
- Tem-se um compartilhamento de atributos e métodos entre classes com base em um relacionamento **hierárquico**
- Herança é um mecanismo **poderoso** em linguagens orientadas a objetos

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

Possibilidades

- Atributos e métodos da superclasse podem ser usados na subclasse diretamente como qualquer outro
  - Atributos e métodos são **herdados**
- Atributos/métodos **adicionais** podem ser declarados na subclasse
- Métodos da superclasse podem ser re-implementados na subclasse
  - Isso é conhecido como **sobreposição** (visto adiante)

Possibilidades

- 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)

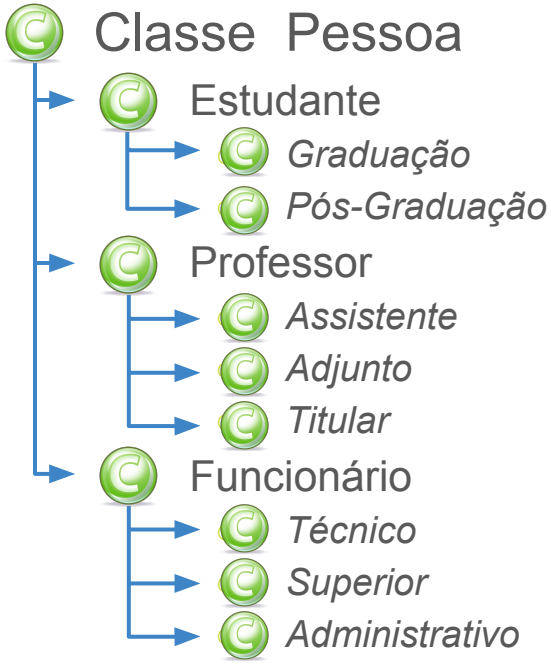
- Métodos muito genéricos podem ser declarados sem implementação
  - Os métodos serão implementados nas subclasses
  - São conhecidos como **métodos abstratos** (visto adiante)

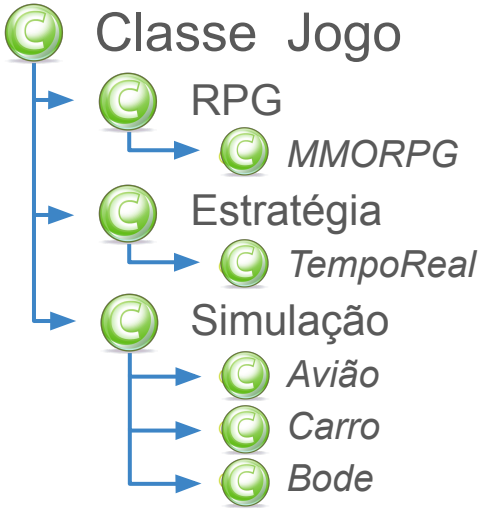
Quando Usar

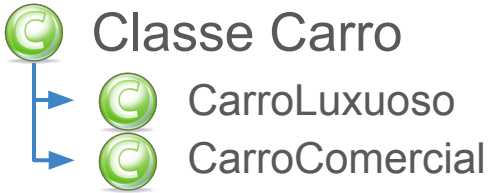
- Herança cria uma relação de “**é um**” entre uma superclasse e a subclasse
  - Se isso não ocorrer, o uso da herança não é válido

- 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`

Exemplos

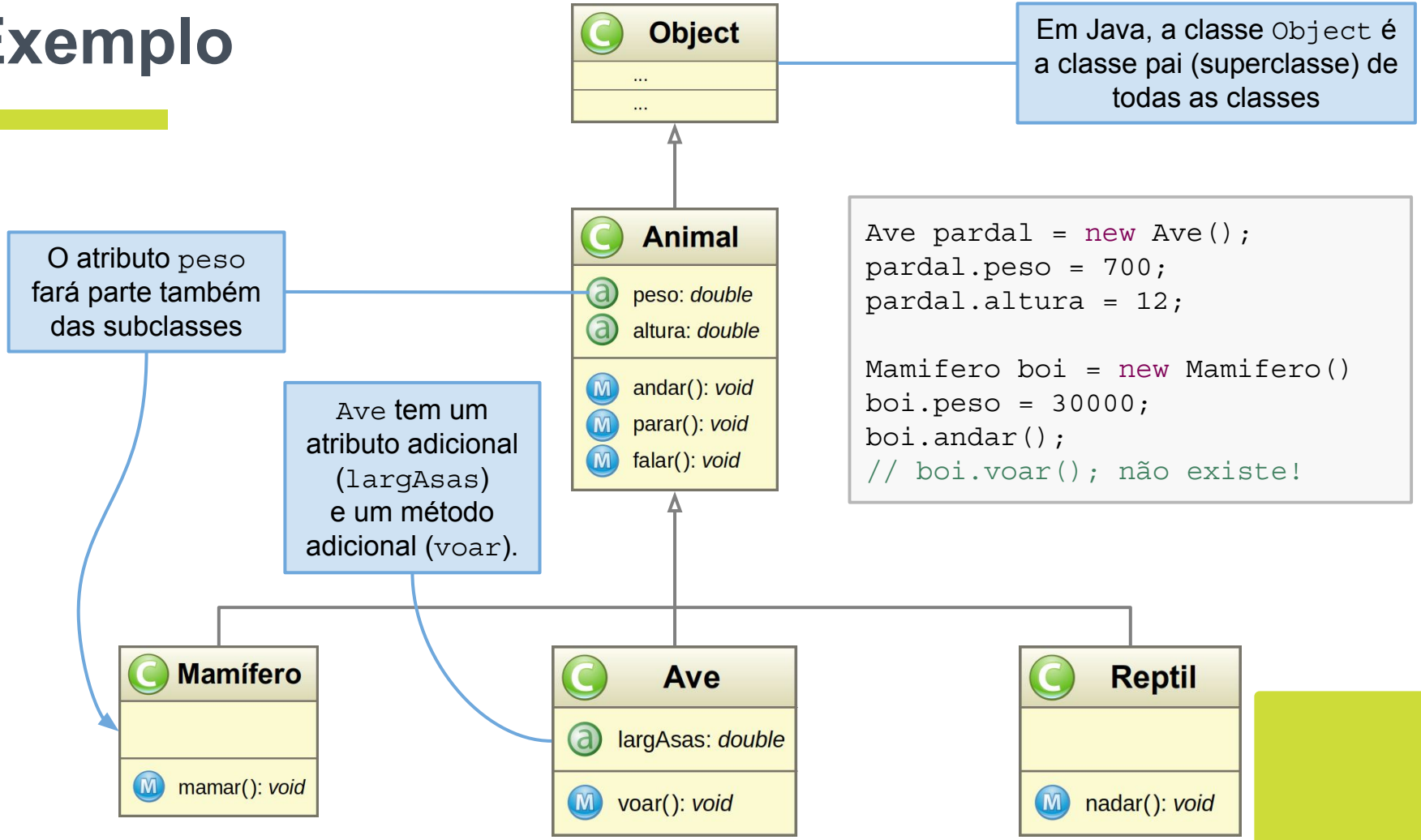

Classe Pessoa
Estudante
Graduação
Pós-Graduação
Professor
Assistente
Adjunto
Titular
Funcionário
Técnico
Superior
Administrativo


Classe Jogo
RPG
MMORPG
Estratégia
TempoReal
Simulação
Avião
Carro
Bode


Classe Carro
CarroLuxuoso
CarroComercial


Classe Conta
Corrente
Poupança
Investimento

Exemplo


Object
...
...
Em Java, a classe Object é a classe pai (superclasse) de todas as classes
Animal
peso: double
altura: double
andar(): void
parar(): void
falar(): void
O atributo peso fará parte também das subclasses
Ave tem um atributo adicional (largAsas) e um método adicional (voar).
Ave pardal = new Ave();
pardal.peso = 700;
pardal.altura = 12;
Mamifero boi = new Mamifero()
boi.peso = 30000;
boi.andar();
// boi.voar(); não existe!
Mamífero
mamar(): void
Ave
largAsas: double
voar(): void
Reptil
nadar(): void

Herança em Java

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

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

Herança Múltipla

- Quando uma **classe herda duas ou mais classes**
- **Java NÃO aceita** herança múltipla
  - Algumas linguagens aceitam herança múltipla: C++, Python, Perl
  - Outras não aceitam também: JavaScript, PHP, Ruby, C#
- Motivos para não aceitar
  - Na natureza, quase não se encontra casos de herança múltipla
    - *Quando encontramos, são exceções, como morcegos (mamíferos que voam) e ornitorrincos (mamíferos ovíparos)*
  - Muitos problemas de implementação:
    - *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()`?*

Classe `Object`

- Forma a raiz da hierarquia de classes em Java
  - 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
  - Faz parte do pacote `java.lang` (importado automaticamente)

- Métodos úteis:

```
String toString()
boolean equals(Object obj)
Object clone()
```

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 ao método `super(…)`
  - Quando **nenhuma chamada ao `super` é feito** no construtor, o Java **automaticamente** inclui a chamada "`super()`" na primeira linha dele
    - *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*

Construtores das Subclasses

```java
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`)

```java
public class Mamifero extends Animal {
  Mamifero(double peso, double altura) {
    super(peso, altura);
  }

  // Método mamar ...
}
```

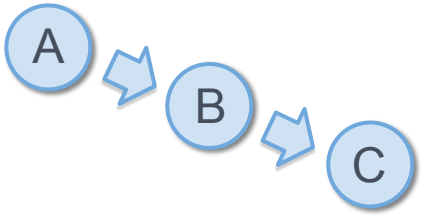
Neste caso, como o `super` foi especificado, o construtor correspondente da superclasse será chamado

```java
public class Ave extends Animal {
  int largAsas;

  Ave(double peso, double altura, int largAsas) {
    super(peso, altura);
    this.largAsas = largAsas;
  }

  // Método voar ...
}
```

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

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);
  }
}
```



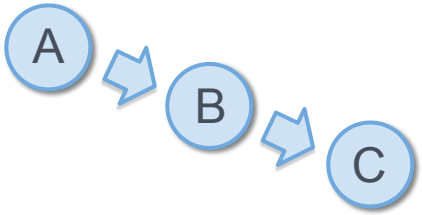

```
class B extends A {
  B() {
    i++;
  }
}
```

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

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

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);
  }
}
```



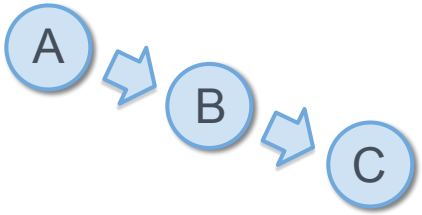

```
class B extends A {
  B() {
    i = 8;
  }
}
```

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

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

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);
  }
}
```



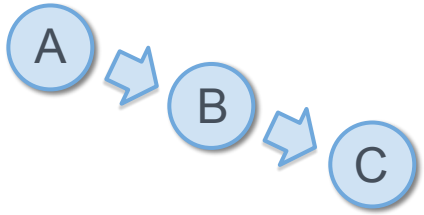

```
class B extends A {
  B() {
    i++;
  }
}
```

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

```
$ javac A.java B.java C.java
$ java A
a.i=1, b.i=2, c.i=3(...) Error (...)
Constructor call must be the first statement in a
constructor (...)
```

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);
  }
}
```

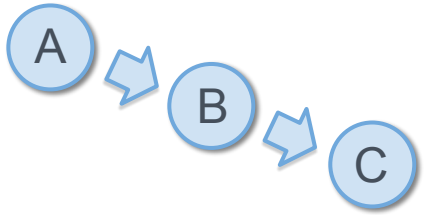



```
class B extends A {
  B() {
    i++;
  }
}
```

```
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
```

Construtores das Subclasses – Quiz 5



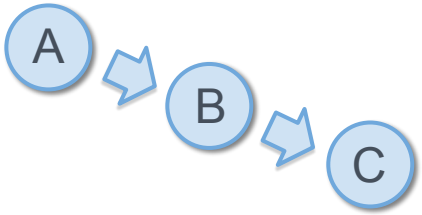

```
class A {
  int i;

  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
```

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++;
  }
}
```

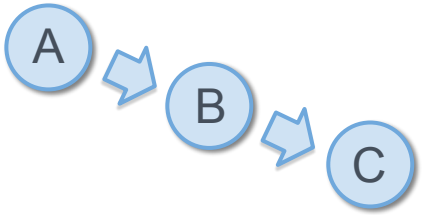
```
class C extends B {
  C() {
    i++;
  }
}
```

```
$ javac A.java B.java C.java
B.java:2: error: constructor A in class A cannot be applied to given types;
  B() {<
  required: int
  found: no arguments
```

super();

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);
  }
}
```



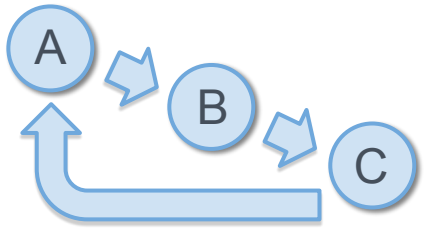

```
class B extends A {
  B() {
    super(6);
    i++;
  }
}
```

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

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

Construtores das Subclasses – Quiz 8

```
class A extends C {
  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);
  }
}
```

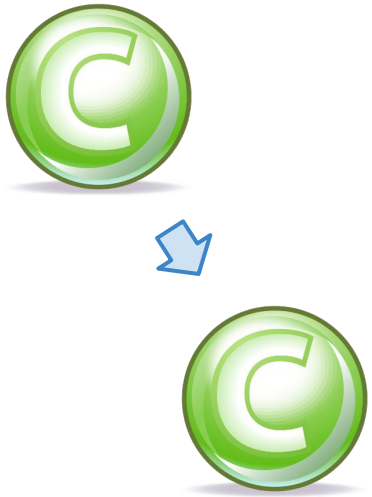



```
class B extends A {
  B() {
    i++;
  }
}
```

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

```
$ javac A.java B.java C.java
A.java:1: error: cyclic inheritance involving A
class A extends C {                              (...)
```

Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Herança
Sobreposição e Generalização

Horácio Fernandes
horacio@icomp.ufam.edu.br

ColabWeb: bit.ly/pp-colabweb

Sobreposição de Métodos

- Na presença de herança, um **método da subclasse** pode ter o **mesmo nome e os mesmos parâmetros** de um método na **superclasse**
- Chamamos isso de **sobreposição**
  - Também conhecido como sobrescrita
  - Também conhecido como redefinição de métodos
  - Em inglês, conhecido como method overriding
- Na sobreposição:
  - Dizemos que o método da subclasse sobrepõe o método da superclasse
  - O método da subclasse pode "**reescrever**" o método da superclasse
    - *Terá uma implementação (comportamento) diferente*

Sobreposição - Exemplo

- Exemplo de sobreposição
  - A classe `Object` declara e implementa o método `equals`:

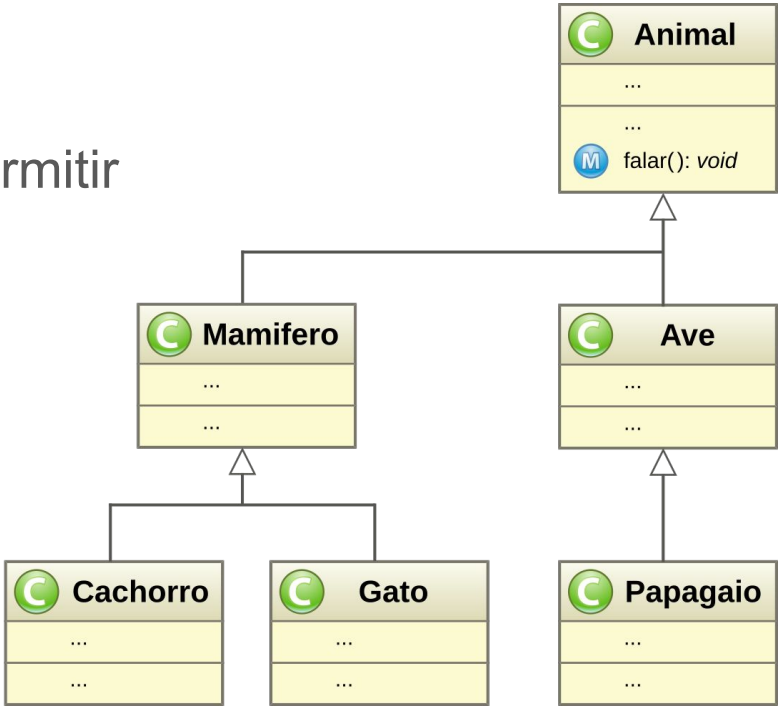
```
public boolean equals(Object obj) {
    return (this == obj);
}
```

  - Entretanto, como visto acima, o método compara apenas referências
  - Na classe `String`, que herda a classe `Object`, o método é sobreposto para comparar conteúdo:

```
public boolean equals(Object anObject) {
    // Implementação dos strings são comparados um a um
}
```

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
  - Animais de classes diferentes, falam de forma diferentes
  - Mas todos os animais falam



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`

Sobreposição de Métodos

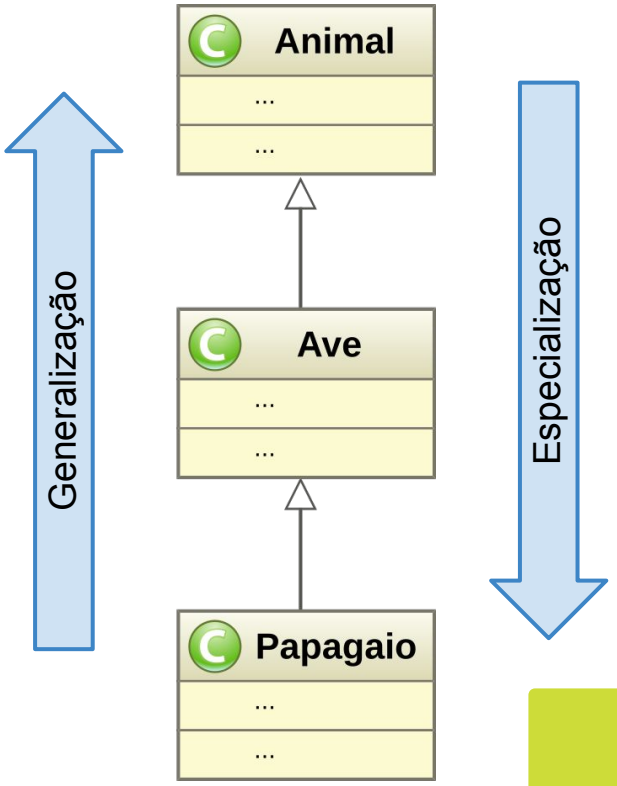
```
Animal    animal =    new Animal();
Cachorro cachorro = new Cachorro();
Gato      gato =      new Gato();
Papagaio papagaio = new Papagaio();

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`

Generalização e Especialização

- **Generalização**
  - trabalhar com classes que identificam características (atributos e métodos) comuns das subclasses
- **Especialização**
  - uso de classes mais específicas (subclasses) com atributos e métodos próprios.



Generalização

- Na generalização:
  - Variáveis que referenciam objetos de uma **superclasse** podem apontar para objetos de **qualquer subclasse**
  - Referenciar objetos de uma subclasse como se fossem de uma superclasse (que é mais genérica), é uma aplicação prática da 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

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 outros métodos

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

OK, pois andar pertence a `Animal`

ERRO! Pois `voar` 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), do inglês, “molde”, “forma”

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

Cast, conversão

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)

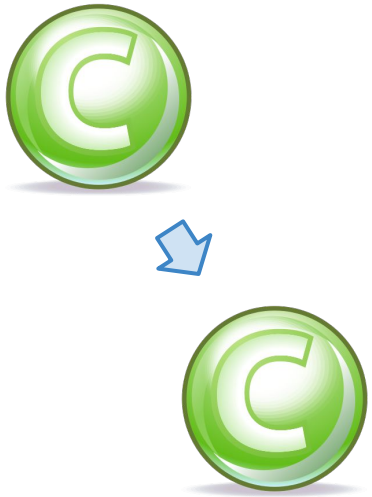
  - Para evitar erro de *cast*, **verifique** antes se o objeto realmente **é uma instância** da classe de destino

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

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

Verdadeiro

Falso

Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Herança
Polimorfismo

ColabWeb: bit.ly/pp-colabweb

Horácio Fernandes
horacio@icomp.ufam.edu.br

Polimorfismo

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

Polimorfismo

- É a capacidade de uma operação (**método**) se **comportar de formas diferentes** dependendo de casos específicos (que indicarão o tipo de polimorfismo).

- O tipo de polimorfismo mais comum é o **polimorfismo por inclusão**.
  - Também conhecido como polimorfismo de **subclasse** (ou **subtipo**)
  - Quando duas ou mais classes relacionadas por herança (e.g., `ClasseA` e `ClasseB`) possuem métodos com a mesma assinatura (nome e parâmetros iguais) mas implementações diferentes (sobreposição de métodos).
  - Ao executar este método em um objeto, a implementação executada irá depender da classe deste objeto. Se este objeto for da `ClasseA`, a implementação presente na `ClasseA` será executada.

Polimorfismo por Inclusão


```
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!
```

Mesmo método (`falar`)

Comportamentos diferentes

Polimorfismo por Inclusão

- Nota de linguagens
  - Em Java, polimorfismo acontece por padrão (sempre)
  - Em C++, para especificar que um método é polimórfico, usa-se a palavra reservada `virtual`
    - *Por isso, estes métodos polimórficos são também conhecidos como métodos virtuais.*

Outros Tipos de Polimorfismo

- Além do **polimorfismo por inclusão**, dependendo de como a diferenciação dos métodos de mesmo nome é feita, tem-se **outros tipos de polimorfismo**:
  - **Polimorfismo por Sobrecarga**
    - *Sobrecarga de métodos também é considerado polimorfismo, uma vez que a execução de um método (pelo nome dele) vai ter comportamento diferente dependendo dos parâmetros.*
    - *Também conhecido como polimorfismo ad-hoc*
  - **Polimorfismo Paramétrico**
    - *Quando um método pode ter um ou mais parâmetros de qualquer tipo. Sua implementação é feita através do uso de **templates** (C++) ou **classes genéricas** (Java).*

Classificação dos Polimorfismos

- Os tipos de polimorfismo podem ser classificados em:
  - **Polimorfismo Estático**
    - *O método que será executado é conhecido e definido em **tempo de compilação**. Por exemplo, no polimorfismo por sobrecarga, o compilador já sabe o método a ser executado*
    - *O compilador é capaz de encontrar possíveis erros antes da execução do programa*
    - *Inclui o Polimorfismo por **Sobrecarga** e o **Paramétrico***
  - **Polimorfismo Dinâmico**
    - *Quando o método a ser executado só pode ser definido em **tempo de execução**.*
    - *É o caso do **Polimorfismo por Inclusão** (subtipo)*
    - *Possíveis erros só são encontrados quando o programa está executando*

Hiding e *Shadowing* de Variáveis

- Dependendo do escopo e do tipo (variáveis locais, argumentos, atributos), **variáveis podem ter nomes iguais**:

```
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);
  }
}
```

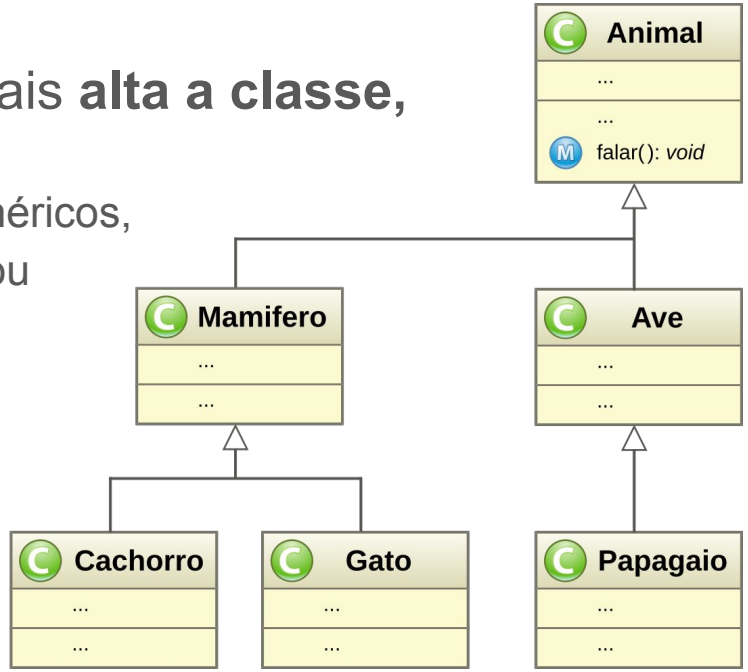
Hiding: um atributo “oculta” outro atributo de mesmo nome presente em uma superclasse.

Shadowing: uma variável “sobreia” outra variável de mesmo nome porque a primeira possui um escopo mais local.

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

Métodos Abstratos

- Em uma hierarquia de classes, quanto mais **alta a classe, mais abstrata** é a sua **definição**
  - Em alguns casos, alguns métodos são 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?
    - *Afinal, todos os animais falam*
- Por isso, em OO é possível definir métodos sem implementá-los!



Métodos Abstratos

- Métodos Abstratos:
  - **Não possuem 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)
}
```

Classes Abstratas

- Classes abstratas **não podem ser instanciadas**!
  - Não é possível criar um objeto de uma classe abstrata
  - Afinal, tais classes possuem métodos sem nenhuma implementação

- 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 sobrepõe todos os métodos abstratos e fornece implementação para cada um deles
  - Caso contrário, ela também deverá ser abstrata

Classes Abstratas

- Exemplo

```
abstract class Animal {
     // Atributos, construtor ...
     abstract void falar();
     // Outros métodos
}
```

```
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

O Modificador “Final”

- Quando aplicado a um **atributo** (visto anteriormente)
  - Indica que o atributo é “**constante**” (não pode ser modificado)

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

- Quando aplicado a um **método**
  - Indica que o método não pode ser **sobreposto** (não pode ser modificado)

```
// Thread.java
public final void setPriority(int newPriority) { ... }
```

- Quando aplicado a uma **classe**
  - Indica que a classe não pode ser **herdada** (não pode ser modificada)

```
public final class String ...
```

Laboratório

- Disponível no ColabWeb
  - bit.ly/pp-colabweb