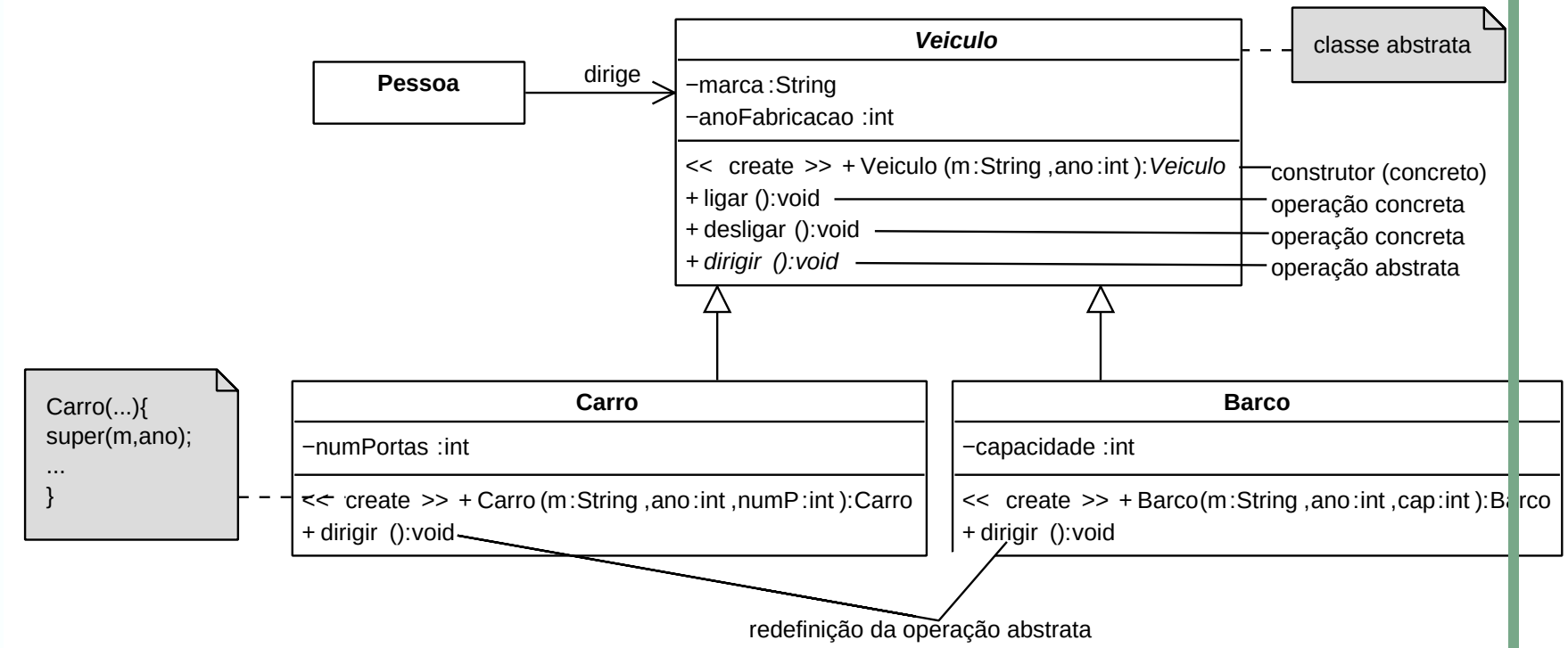
Classes Abstratas (I)

- **Classe abstrata** é uma classe que não tem instâncias diretas, mas cujas descendentes podem ter
- **Classe concreta** é uma classe que pode ser instanciada
- Uma classe abstrata pode definir o protocolo para uma operação sem fornecer o código correspondente
- Uma operação é chamada **abstrata** quando ela não tem implementação

Classes Abstratas (II)



Classes e Operações Abstratas (I)

Em C++, uma classe abstrata é criada com a presença de **pelo menos** uma operação abstrata (pure virtual)

```
public:
        virtual void operacaoAbstrata( ...) = 0;
```

- Toda operação abstrata utiliza acoplamento dinâmico
- Em Java, uma operação abstrata é definida através do modificador *abstract*

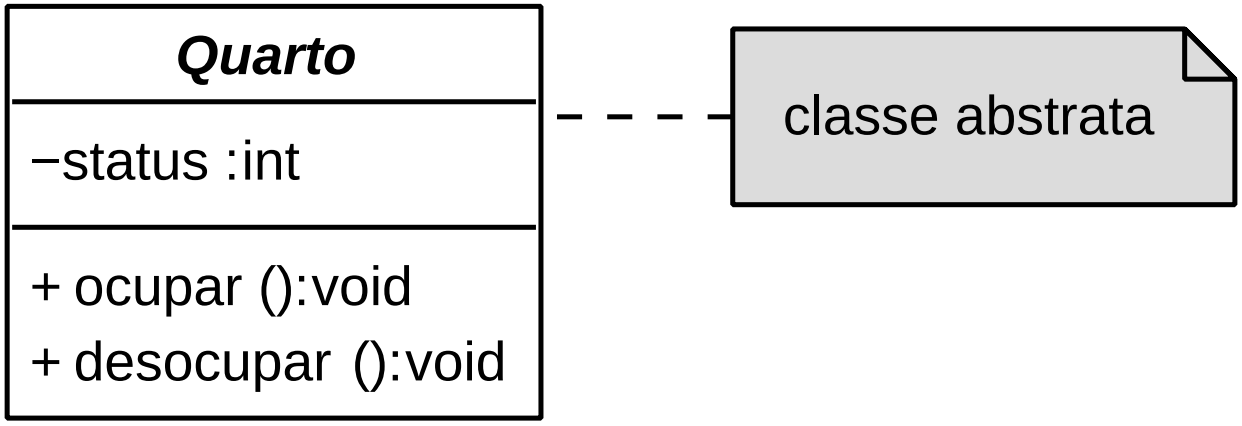
Classes e Operações Abstratas (II)

- A noção de classe abstrata abrange duas idéias:
  1. Impossibilidade de haver instâncias diretas (obrigatoriamente)
  2. Presença de operações abstratas (não necessariamente)

Classes e Operações Abstratas (III)

Em Java, é possível definir uma classe abstrata **sem** nenhuma operação abstrata usando o modificador *abstract* na frente da classe:

```
abstract class Quarto{
public void ocupar(); ...}
```

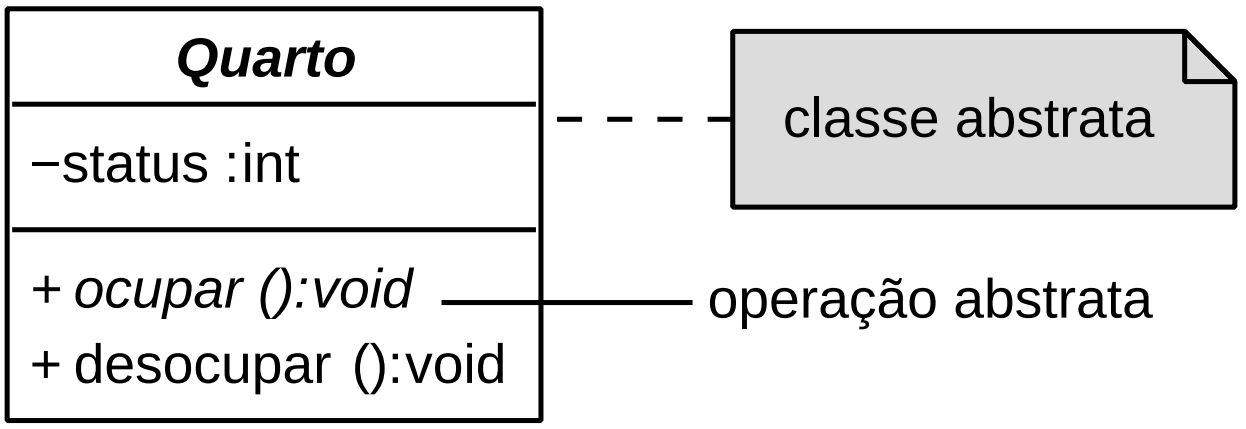


Classes e Operações Abstratas (IV)

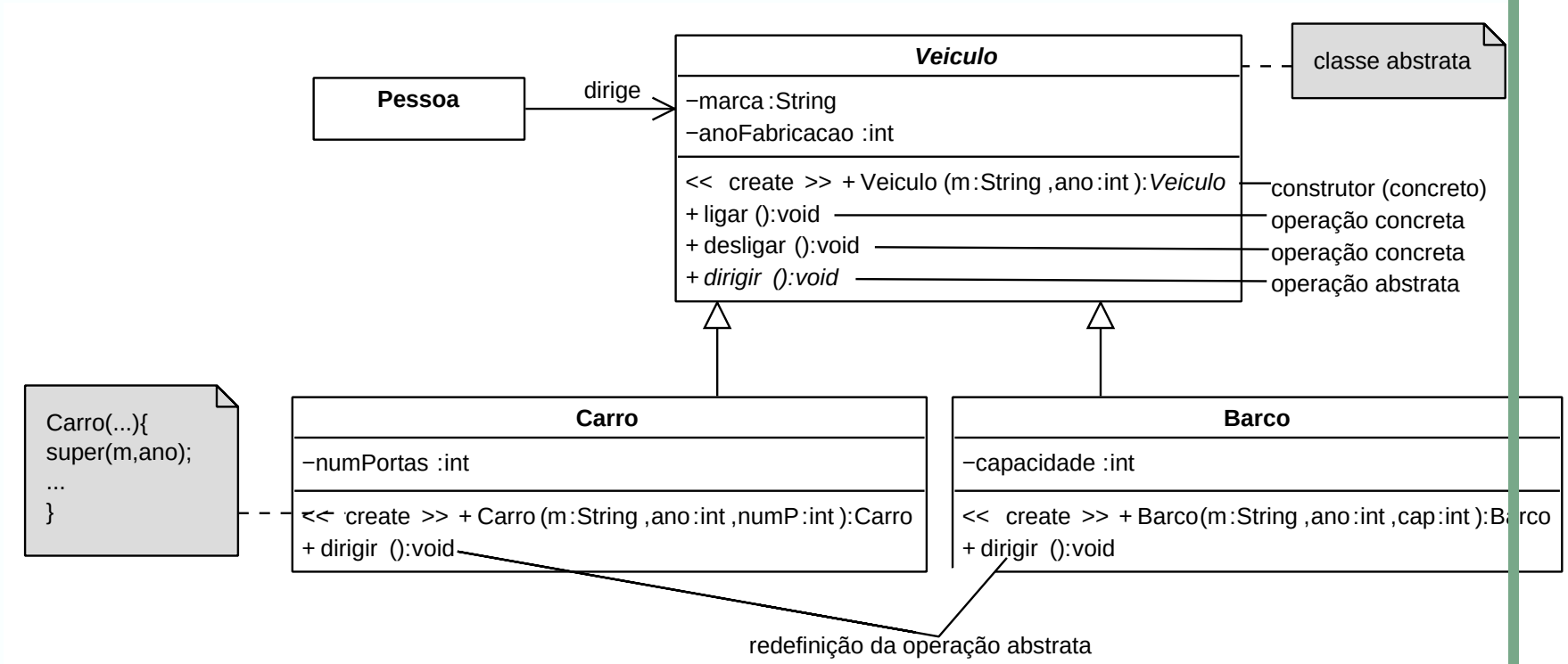
Em Java, a presença de uma operação abstrata numa classe obriga que a classe seja explicitamente declarada como abstrata:

```
abstract class Quarto{
  public abstract void ocupar(); ...}
```

Em C++, uma classe abstrata **exige** a presença de pelo menos uma operação abstrata, **não** existindo a opção de poder ser declarada explicitamente como abstrata.



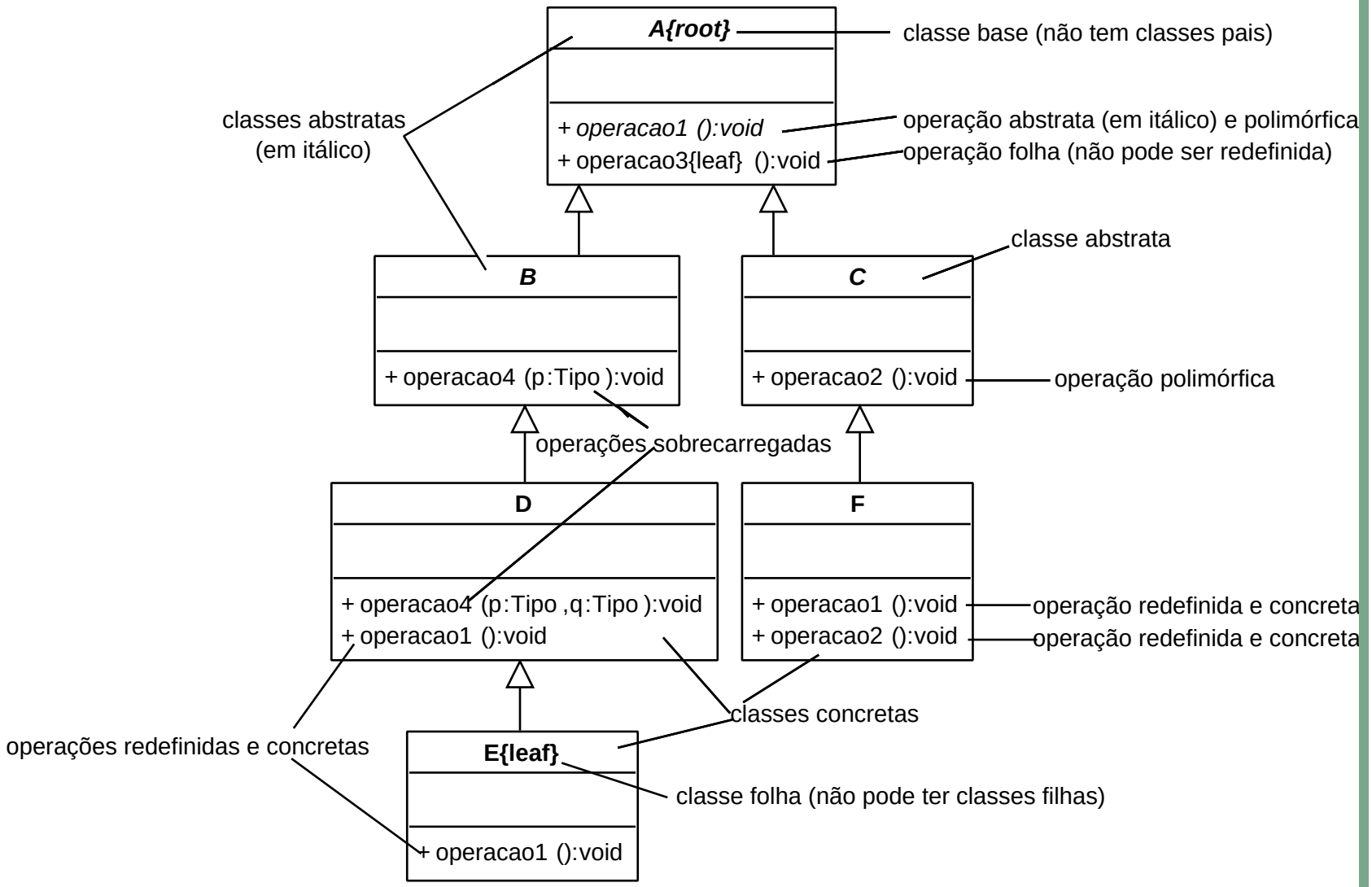
Classes e Operações Abstratas (V)



Classes e Operações Abstratas (VI)

- A classe Veiculo é abstrata, isso significa que os objetos que as pessoas dirigem são instâncias das classes concretas Carro e Barco
- Quando uma pessoa chama a operação *dirigir()* para uma variável do tipo Veiculo, o efeito obtido depende do tipo do objeto que é referenciado na ocasião
- Se for um objeto da classe Barco, a operação *dirigir()* aciona a hélice do barco usando a implementação especificada na classe Barco
- Se for um objeto da classe Carro, a operação *dirigir()* aciona o motor que faz as rodas do carro moverem usando a implementação especificada na classe Carro

Classes e Operações Abstratas (VII)



Classes e Operações Abstratas (VIII)

```
public abstract class A {
    public abstract void operacao1();
    public final void operacao3() { ... }
 } //fim da classe A

public abstract class B extends A {
    public void operacao4(Tipo p) {...}
 }// fim da classe B

public abstract class C extends A {
    public void operacao2() {...}
 }// fim da classe C
```

```
public class D extends B {
    public void operacao4(Tipo p, Tipo q) {...}
    public void operacao1() {...}
 }// fim da classe D

public final class E extends D {
    public void operacao1() {...}
 }//fim da classe E

public class F extends C {
    public void operacao1() {...}
    public void operacao2() {...}
 }// fim da classe F
```

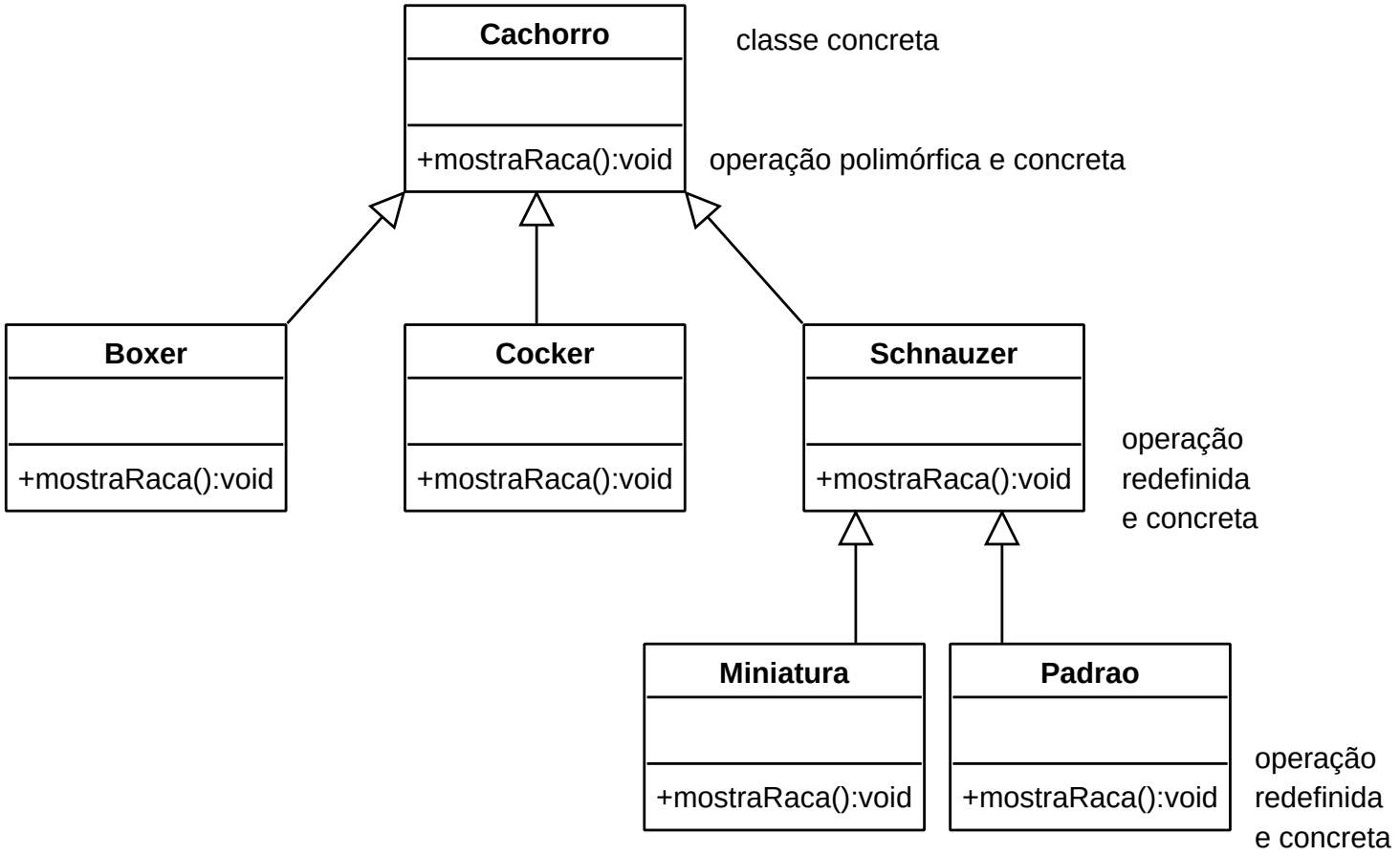
Classes e Operações Abstratas (IX)

- Em UML, o nome de uma classe (ou operação abstrata) é escrito em *itálico*.
- As classes A, B e C são abstratas. Nenhuma delas pode gerar instâncias diretas.
- As classes D, E e F são concretas, isto é, elas podem gerar instâncias através do uso de seus construtores (por exemplo, *new D()*).
- Uma classe *folha* ("leaf") não tem classes derivadas (equivale ao modificador de *final* Java aplicado à classe).

Classes e Operações Abstratas (X)

- Uma classe *raiz* ("root") não tem classe base. Ela não é derivada de nenhuma classe do sistema e indica o início de uma hierarquia de classes. Exemplo: classe Object em Java
- Uma operação *folha* ("leaf") é considerada não polimórfica e não pode ser redefinida em lugar nenhum da hierarquia (equivale ao modificador *final* de Java aplicado a uma operação)
- Em UML, uma operação é polimórfica por definição; significando que é possível redefinir operações com a mesma assinatura em diversos pontos da hierarquia de classes (mesma semântica de Java)

Exemplo: Hierarquia de Classes para Raça de Cães (I)



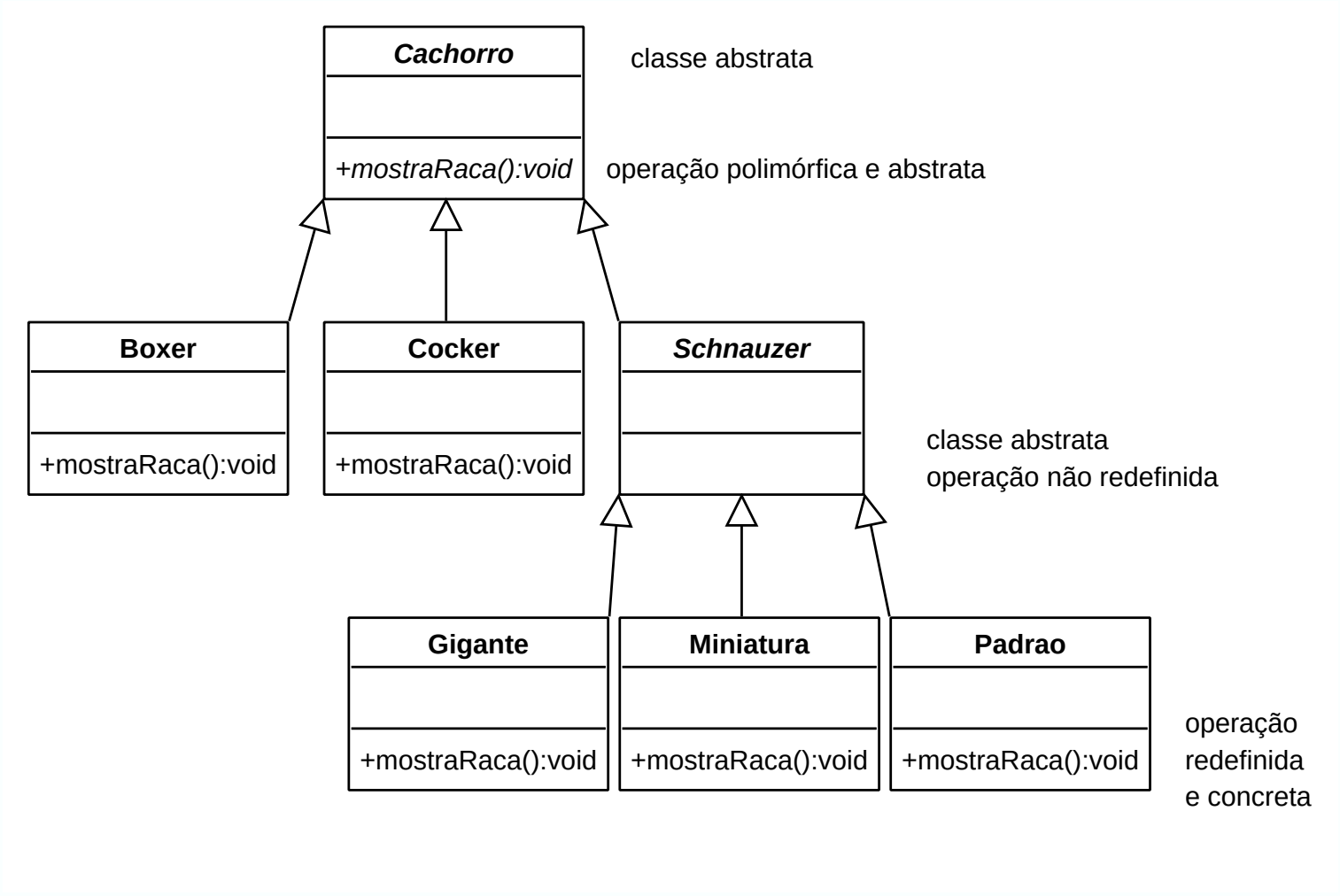
Exemplo: Hierarquia de Classes para Raça de Cães (II)

```
class Cachorro{
public void mostraRaca(){
System.out.println(‘‘Raça não definida’’);
}}
class Schnauzer extends Cachorro{
public void mostraRaca(){
System.out.println(‘‘Raça Schnauzer’’);
}}
class Miniatura extends Schnauzer{
public void mostraRaca(){
System.out.println(‘‘Raça Schnauzer
miniatura’’);
}}
```

Exemplo: Hierarquia de Classes para Raça de Cães (III)

```
Cachorro apCach; // referência para objetos do tipo
                 //Cachorro e seus subtipos
apCach = new Schnauzer();
apCach.mostraRaca(); // imprime ‘‘Raça Schnauzer’’
apCach = new Miniatura();
apCach.mostraRaca(); // imprime ‘‘Raça
                       // Schnauzer miniatura’’
apCach = new Cachorro();
apCach.mostraRaca(); // imprime ‘‘Raça não definida’’
```

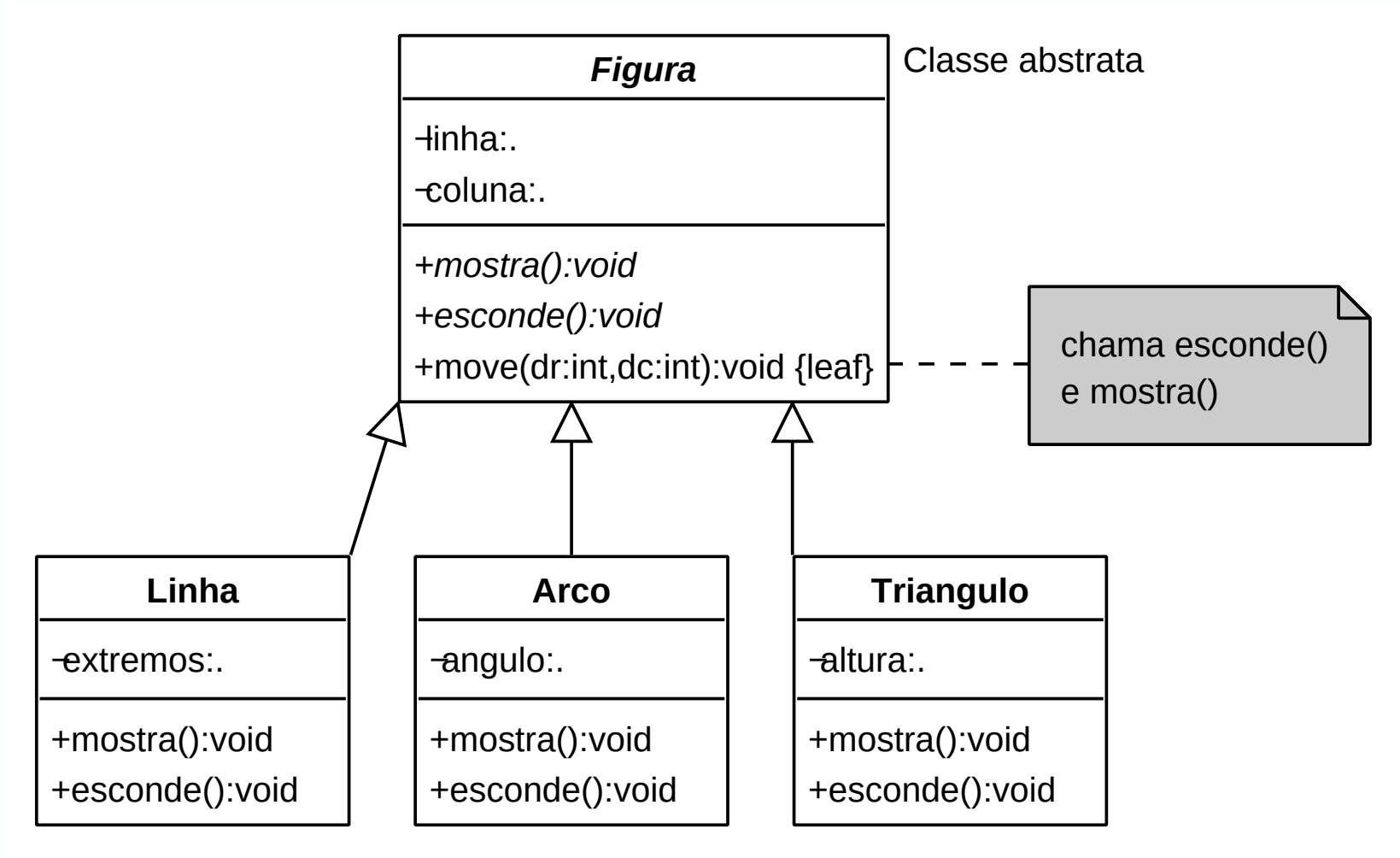
Hierarquia de Raças Modificada (I)



Hierarquia de Raças Modificada (II)

```
Cachorro apCach;
apCach = new Cachorro(); // ERRO: instanciacao
                         // de classe abstrata
apCach = new Schnauzer(); // ERRO: instanciacao
                          // de classe abstrata
apCach = new Miniatura(); // OK: instanciacao
                          // de classe concreta
apCach.mostraRaca(); // imprime ‘‘Raça
                     // Schnauzer miniatura’’
apCach = new Boxer(); // OK: instanciacao
                      // de classe concreta
apCach.mostraRaca(); // imprime ‘‘Raça
                     // Boxer’’
```

Exemplo: Figuras Gráficas (I)



Exemplo: Figuras Gráficas (II)

```
abstract class Figura{
   private int linha;
   private int coluna;
   public Figura(int r, int c){ // construtor
      linha = r; coluna = c;}
   public abstract void mostra(); // op. abstrata
   public abstract void esconde(); // op. abstrata
   public final void move(int dr, int dc){
      this.esconde();
      linha += dr;
      coluna += dc;
      this.mostra();
   }
} // fim da classe Figura
```

Exemplo: Figuras Gráficas (III)

- Note que move() é declarada como "leaf", e, portanto, implementa acoplamento estático.
- Esse método representa um **ponto de congelamento** ("frozen spot") da hierarquia de classes. Isto é, ele não pode ser redefinido em lugar nenhum, pois a operação não pode ser redefinida.
- O método move() chama dois outros métodos polimórficos (passivéis de serem redefinidos) para executar ações dependentes de cada tipo de objeto (Linha, Arco e Triângulo).
- Os métodos mostra() e esconde() são chamados de **pontos de adaptação** ("hot spots") da hierarquia de classes, pois são polimórficos (e abstratos) e podem ser redefinidos em diversos pontos da hierarquia.