

Curso de Java

Classes Abstratas, Exceções e Interfaces





Classes Abstratas

- Classes Abstratas
 - construção de uma classe abstrata
 - construção de classes derivadas



Classes e Herança

- Uma classe define um conjunto de dados um conjunto de métodos
- Todos os objetos de uma classe mantêm o mesmo conjunto de atributos e métodos.
- Através do mecanismo de herança de tendo definido uma *classe base* é possível criar *classes derivadas* que
 - herdam os atributos e métodos da classe base
 - definem novos atributos e métodos
 - podem redefinir os métodos herdados



Classe Abstrata

- Uma classe abstrata em Java define atributos e métodos.
- Numa classe abstrata, um método pode ser definido com o modificador 'abstract'. Nesse caso
 - a classe abstrata não implementa os método abstratos.
 - As classes derivadas devem implementar os métodos abstratos.



Um exemplo

- Suponha uma aplicação que deve manipular figuras geométricas como retângulos, triângulos, círculos, etc.
- Uma classe Figura, abstrata, pode ser usada para
 - definir os atributos comuns a todas as figuras a serem tratadas
 - servir como *classe base* para as classes que descrevem as demais figuras



Um exemplo: a *classe mãe*

```
public abstract class Figura
{
    public int x0;
    public int y0;

    public Figura() { x0 = 0; y0 = 0; }

    public Figura(int x, int y){ x0 = x; y0 = y; }

    public String toString(){
        return "Figura("+x0+" : "+ y0+")";
    }

    public abstract int area();

    public abstract int perimetro();
}
```



Um exemplo: a *classe base*

- Nesse exemplo

- os métodos

```
public Figura() { x0 = 0; y0 = 0; }  
public Figura(int x, int y){ x0 = x; y0 = y; }
```

são os construtores para os objetos da classe **Figura**, utilizados para a criação de objetos dessa classe.

- têm o mesmo nome que a classe
- a diferença entre eles está na lista dos parâmetros

Em Java é possível ter mais de um método com o mesmo nome, desde que as listas de parâmetros sejam diferentes quanto ao número ou quanto ao tipo dos parâmetros.



Um exemplo: a *classe base*

- Nesse exemplo
 - os atributos

```
public int x0;  
public int y0;
```

- são utilizados para indicar as coordenadas do 'ponto origem' da figura.
- o modificador **public** indica que eles podem ser acessados por outras classes que façam uso de objetos da classe **Figura**.



Um exemplo: a *classe derivada*

- A partir da classe podemos criar, por exemplo, uma classe que descreve um retângulo que
- acrescenta novos atributos, indicando a altura e largura do retângulo.
 - redefine os métodos `area()`, `perimetro()` e `toString()`.



Um exemplo: a *classe derivada*

```
public class Retangulo extends Figura {
    int largura, altura;
    public Retangulo(int b, int a) {
        super(); largura = b; altura = a;
    }

    public Retangulo (int x, int y, int b, int a){
        super(x,y); largura = b; altura = a;
    }

    public String toString(){
        return ("Retangulo("+x0+": "+y0+": " +
            largura+": "+altura+") ");
    }

    public int area(){ return altura*largura; }
    public int perimetro() { return (altura+largura)*2; }
}
```



Um exemplo: a *classe derivada*

- Nesse exemplo
 - a classe Retângulo é definida como classe derivada da classe Figura. A herança é indicada na declaração da classe:

```
public class Retangulo extends Figura
```

- os atributos

```
int altura, largura;
```

definem a altura e a largura de cada objeto da classe **Retângulo**, que também herdam os atributos **x0** e **y0** da classe **Figura**.



Um exemplo: a *classe derivada*

Nesse exemplo

- o construtor

```
public Retangulo(int b, int a){  
    super(); largura = b; altura = a;  
}
```

indica que o construtor **Figura()** da *classe base* deve ser chamado antes da execução do mesmo (é importante que seja antes).

- Um construtor sem parâmetros, em Java é um construtor padrão (*default*) e será declarado implicitamente se não for explicitamente declarado na classe.
- A chamada ao construtor padrão da *classe mãe* também é implícita se não for indicada pelos construtores das *classes derivadas*. Isso significa que o uso de **super()** feita no exemplo acima não é necessária.



Um exemplo: a *classe derivada*

- Nesse exemplo

- o construtor

```
public Retangulo (int x, int y, int b, int a){  
    super(x,y);  
    largura = b;  
    altura = a;  
}
```

- indica que o construtor *Figura(x,y)* da *classe base* deve ser chamado antes da execução do mesmo através do comando **super (x, y)** na declaração do método.
 - neste caso, como o construtor da classe base a ser chamado não é o construtor padrão, a indicação deve ser explícita.



Exemplo de uso

```
public class Teste {  
  
    static void teste(Figura f, String s){  
        System.out.println(s+" ==> "+f.toString()+  
            " area:"+f.area() +  
            " perimetro:"+f.perimetro());  
    }  
  
    public static void main(String[] args) {  
        Retangulo r1 = new Retangulo(1,2,5,10);  
        Retangulo r2 = new Retangulo(5,10);  
        teste(r1, "r1");  
        teste(r2, "r2");  
    }  
}
```



Outro exemplo

A classe **Retangulo** pode ser usada como base para criar uma classe derivada, por exemplo a classe **Quadrado**:

- um quadrado é basicamente um retângulo que tem a altura igual à largura.

A seguir é mostrada a classe **Quadrado**, derivada de **Retangulo**.



Outro exemplo

```
public class Quadrado extends Retangulo {  
  
    public Quadrado(int a) { super(a,a); }  
  
    public Quadrado (int x, int y,int a) { super(x,y,a,a); }  
  
    public String toString(){  
        return ("Quadrado (" + x0 + " : " + y0 + " : " + altura + ")");  
    }  
}
```




Outro exemplo

- Nesse exemplo,
 - a classe Quadrado é derivada de Retangulo, que por sua vez é derivada de Figura.
 - apenas o método `toString()` foi redefinido.
 - os construtores para Quadrado se limitam a chamar adequadamente o construtor da *classe base*:

```
public Quadrado(int a){ super(a,a); }
```

```
public Quadrado (int x, int y,int a){ super(x,y,a,a); }
```



A classe Object

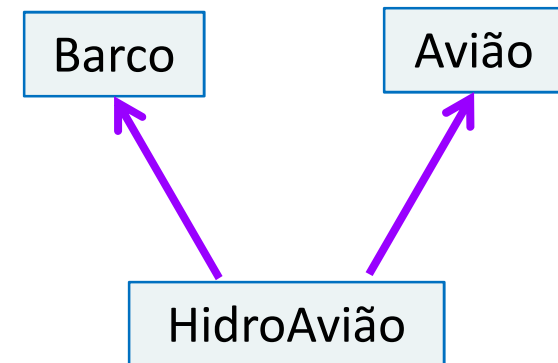
A classe **Object** é pré-definida em Java

- Quando definimos uma nova classe que não é derivada de uma classe base, essa classe é implicitamente derivada de **Object**.
- A classe **Object** é portanto a classe base da qual todas as classes em Java são derivadas.



Herança múltipla

- *Herança múltipla* ocorre quando uma classe é derivada de mais de uma classe base, herdando a união dos métodos e atributos.
- Java não permite *herança múltipla*.





Interfaces

- Uma classe abstrata, além de definir métodos abstratos, pode implementar alguns métodos.
- Uma *interface* define apenas um conjunto de métodos abstratos.
- Uma classe pode *implementar* mais de uma interface.
- A *implementação* de uma ou mais interfaces não exclui a possibilidade de *herança*.
- O conceito de *polimorfismo* também é aplicável às *interfaces* implementadas por uma classe.



Interfaces – um exemplo

```
public interface Ordenavel {  
    public boolean precede(Ordenavel x);  
}
```

```
public class Retangulo extends Figura implements Ordenavel{  
    int largura, altura;  
    ...  
    public int area(){ return altura*largura; }  
    ...  
    public boolean precede(Ordenavel x){  
        return this.area() <= ((Figura)x).area();  
    }  
}
```



Interfaces – um exemplo

- Neste exemplo
 - A classe Retangulo, derivada de Figura, implementa a interface Ordenavel.
 - O método Retangulo.precede() faz um *casting* do parâmetro x para Figura.
 - A referência this está sendo usada para definir o objeto ao qual o método precede() é aplicado. Essa referência é necessária em alguns casos para evitar ambiguidades (neste caso em particular pode ser eliminada).



Interfaces – outro exemplo

```
public interface ComandosAviao {  
    ...  
    public void subir();  
}
```

```
public interface ComandosBarco {  
    ...  
    public void ancorar();  
}
```

```
public class HidroAviao extends Transporte  
    implements ComandosAviao, ComandosBarco {  
    ...  
    public void subir()    { /** implementação **/ }  
    public void ancorar() { /** implementação **/ }  
    ...  
}
```



Exceções

- Um programa sempre está sujeito a situações não previstas que podem levar a erros em tempo de execução:
 - Tamanho do vetor excedido
 - Arquivo de entrada não encontrado
 - Overflow
 - *casting* inválido (p. ex. em Figura.precede()).
 - etc.
- Em C, esse tipo de situação leva à interrupção do programa (e muitas vezes a 'segmentation fault').



Exceções

- Em Java, o programa tem condições de assumir o controle de execução quando ocorre uma situação de erro não prevista.
- Isso é feito através do mecanismo de *tratamento de exceções*:
 - ao detectar a situação de erro a máquina virtual (JVM) gera uma exceção.
 - se o programa em execução tiver um tratador de exceções, este assume o controle da execução.
 - se a exceção não tiver um tratador associado, o programa é interrompido e a JVM gera uma mensagem de erro.



Tratamento de exceções

- Um tratador é associado a uma seqüência de comandos delimitada por 'try{...}catch'.
- Se ocorrer uma exceção num dos comandos protegidos pelo tratador, este é ativado: os comandos de tratamento da exceção são executados.



Tratamento de Exceções

- Estrutura geral:

```
public class Retangulo extends Figura implements Ordenavel{
    int largura, altura;
    ...
    try{
        ...
        <meus comandos>
        ...
    }catch ( <exceção> ee ) {
        ...
        <comandos de tratamento da exceção>
        ...
    }
}
```



Exceções: um exemplo

```
public class Retangulo extends Figura implements Ordenavel{
    ...
    public boolean precede(Ordenavel x){
        try {
            return this.area() <= ((Figura)x).area();
        } catch (ClassCastException ee) {
            System.out.println("'casting' inválido para Figura");
            return false;
        }
    }
    ...
}
```



Exceções

- Java define a classe Exception, da qual podem ser derivadas outras classes.
- A linguagem define uma hierarquia de classes derivada de Exception (a exceção ClassCastException faz parte dessa hierarquia).
- O programador pode definir suas próprias exceções.

```
public class MyException extends Exception {  
    ...  
}
```



Exceções

- Um método pode gerar uma exceção através do comando throw.
- Nesse caso, a declaração do método deve indicar a geração da exceção. Um exemplo:

```
public class Retangulo extends Figura implements Ordenavel{
    ...
    public boolean precede(Ordenavel x) throws myException{
        if(! x instanceof Figura) throw new myException();
        return this.area() <= ((Figura)x).area();
    }
    ...
}
```



Outro exemplo

- O próximo exemplo mostra a implementação de uma fila de prioridades.
 - a fila é formada por objetos de classes que implementam a interface Sortable, que prevê o método precedes(Sortable x).
 - operações implementadas pela fila
 - insert(Sortable x)
 - Sortable remove()



A interface Sortable

- A interface **Sortable** será feita como parte do pacote **prqueue**, que contém a classe **PrQueue**.

```
package prqueue;  
  
public interface Sortable {  
    public boolean precedes(Sortable x);  
  
}
```




A classe PrQueue

```
package prqueue;

public final class PrQueue {
    private int maxSize,idx;  /** tamanho máximo da fila
    **/
    private Sortable queue[]; /** vetor que implementa a
    fila **/

    public PrQueue(int size){ ... } /** construtor **/
    public int size() { ... }      /** tamanho da fila **/
    public void insert(Sortable x) throws Exception { ... }
    public Sortable remove() throws Exception { ... }
    private void downHeap( int m) { ... }
    private void upHeap(int m) { ... }

}
```



O método insert()

```
public void insert(Sortable x) throws Exception {
    if((idx+1) >= maxSize)
        throw new Exception("PrQueue: overflow");
    queue[++idx] = x;
    upHeap(idx);
}

private void upHeap(int m) {
    int r = 0;
    Sortable x = queue[m];
    int j = m/2;
    while((m > 0) && (j >= r) && (x.precedes(queue[j]))) {
        queue[m] = queue[j];
        m = j;
        j = j/2;
    }
    queue[m] = x;
}
```



O método remove()

```
public Sortable remove() throws Exception {
    if(idx == -1) throw new Exception("PrQueue: empty");
    Sortable x = queue[0]; queue[0] = queue[idx--];
    downHeap(idx);
    return x;
}

private void downHeap( int m) {
    int r = 0;
    Sortable x = queue[r];
    while (2*r < m) {
        int f = 2*r+1;
        if ((f < m) && queue[f+1].precedes(queue[f])) ++f;
        if (x.precedes(queue[f])) break;
        queue[r] = queue[f];
        r = f;
    }
    queue[r] = x;
}
```



A classe PrQueue

- Neste exemplo
 - a classe PrQueue foi declarada como final. Isso impede a criação de classes derivadas de PrQueue.
 - os métodos insert() e remove() podem gerar exceções.
 - os objetos da fila podem ser quaisquer objetos de classes que implementem a interface Sortable.



Exemplo de uso

```
package prqueue;

public class Test1 {
    public static void main(String[] args) throws Exception {
        PrQueue pq = new PrQueue(args.length);
        for(int i = 0; i < args.length; i++)
            pq.insert(new XStr(args[i]));

        for(int i = 0; i < txt.length; i++){
            String s = ((Sortable) pq.remove()).toString();
            System.out.println("s: "+s);
        }
    }
}
```



Exemplo de uso

- Neste exemplo
 - os objetos inseridos na fila são construídos a partir dos parâmetros passados na linha de comando (String [] args)
 - a classe xStr deve implementar a interface Sortable.
 - como os métodos PrQueue.insert() e PrQueue.remove() podem gerar exceções, é necessário que o seu uso ocorra dentro de um tratador ou dentro de um método que gere exceções.



A classe xStr

```
class xStr implements Sortable{
    private String str;

    public xStr(String str){
        this.str = str;
    }

    public String toString() { return str; }

    public boolean precedes(Sortable x){
        if((x == null) || (str == null))return false;
        return str.compareTo(((xStr)x).str) < 0;
    }
}
```



Outro exemplo

- O exemplo a seguir mostra o uso da classe PrQueue para implementar o algoritmo de Dijkstra, para determinar os caminhos mínimos num grafo, a partir de um dado vértice.
- Nesse exemplo são criadas as classes
 - Graph: implementa o grafo e o método minPaths().
 - Vertex : descreve os vértices do grafo e implementa a interface Sortable (classe interna a Graph).
 - Edge: descreve as arestas do grafo (classe interna a Graph).



A classe Graph

```
package graph;
import prqueue.*;

public class Graph {
    private Vertex nodes[];
    public Graph(int n){ ... }
    int size() { return nodes.length; }
    public void addEdge(Vertex v1, Vertex v2, int d ) { ... }
    public void addEdge(int v1, int v2, int d) { ... }
    public void minPaths(Vertex v) throws Exception { ... }
}
```



O método minPaths()

```
public void minPaths(Vertex v) throws Exception {  
  
    /** iniciar a fila de prioridades **/  
    PriorityQueue pq = new PriorityQueue(size());  
    Edge e = v.getAdjList();  
    while(e != null){  
        Vertex w = e.getEnd();  
        pq.insert(w);  
        w.setDist(e.getDist());  
        e = e.getNext();  
    }  
  
    /** <continua > **/  
}
```



O método minPaths()

```
/** <continuação> */  
while(pq.size() > 0){  
    Vertex w = (Vertex)pq.remove();  
    e = w.getAdjList();  
    while(e != null){  
        Vertex x = e.getEnd();  
        int nd = w.getDist() + e.getDist();  
        if(nd < x.getDist()){  
            x.setDist(nd);  
            x.setPrev(w);  
            pq.insert(x);  
        }  
        e = e.getNext();  
    }  
}  
}
```



A fila de prioridades

- A fila de prioridades utilizada nesta aplicação deve ter a seguinte característica:
 - Como um vértice pode ter a sua distância atualizada diversas vezes durante o processo, é necessário garantir que apenas uma referência ao mesmo fique na fila.
 - O método insert() deve garantir essa condição.
 - A fila de prioridades descrita anteriormente se propõe a ser genérica e não atende a essa condição.



A classe Vertex

```
class Vertex implements Sortable{
    public static final int MAX = 99999999;
    private String name;          /** nome do vértice      **/
    private int dist;             /** distância à origem **/
    private Vertex prev;          /** vertice anterior  **/
    private Edge adjList;         /** lista de adjacência **/
    public Vertex(String n){ ... } /** construtor      **/
    public int getDist()          { return dist; }
    public Vertex getPrev()       { return prev; }
    public String getName()       { return name; }
    public Edge getAdjList()      { return adjList; }
    public void setDist(int d)    { dist = d; }
    public void setPrev(Vertex p) { prev = p; }
    public void addEdge(Edge e)   { ... }
    public boolean precedes(Sortable n){
        return this.dist < ((Vertex)n).dist;
    }
}
```

A classe Edge

```
class Edge{
    private Vertex end;    /** vértice 'final' **/
    private int dist;      /** distância      **/
    private Edge next;     /** próxima aresta **/

    public Edge(Vertex e, int d, Edge n){
        end = e;
        dist = d;
        next = n;
    }

    public Vertex  getEnd() { return end;    }
    public int     getDist() { return dist;  }
    public Edge    getNext() { return next;  }
    public void    setNext(Edge e) { next = e; }
}
```