

POO: Relacionamento entre classes e Herança


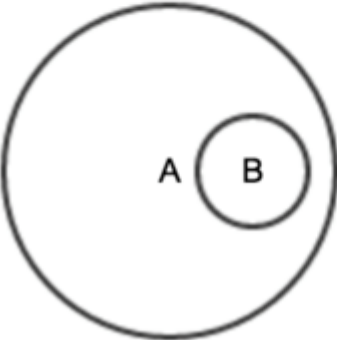
Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Relacionamento entre classes

- Quando várias classes representam um sistema e interagem entre si, podemos ter relacionamentos ou dependências, ou ambos, entre estas classes.
- Um **relacionamento** de uma **classe A** para uma **classe B** ocorre quando, na **definição da classe A** for necessária a **definição da classe B**.
- Um relacionamento pode ser classificado em duas grandes categorias:
 - **Em relação a sua semântica (significado): associação ou herança**
 - **Em relação a sua multiplicidade: 1:N (um-para-muitos)**

<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#repetindo-codigo>

Relacionamento semântico

Diagrama Relacional	Semântica
 <p>Associação</p>	<p>A classe A pode se relacionar com a classe B e/ou vice-versa. Ou seja, podemos colocar A como um atributo de B ou B como um atributo de A.</p> <pre>public class A{ B b ; public A(B b){ this.b=b; } } ou public class B { A a ; public B(A a){ this.a=a; } }</pre>
 <p>Herança</p>	<p>Tipo especial de associação do tipo generalização-especialização, onde B é uma versão mais especializada da classe A, mais genérica.</p>

Exercício relacionamento semântico: associação

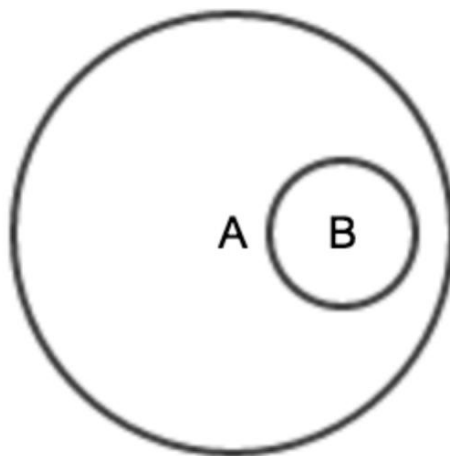
Imagine que você esteja desenvolvendo um jogo e em determinado fase do desenvolvimento houve a necessidade de modelar e verificar, via programação, se um ponto (x,y) está dentro de um circunferência (centro (x,y) , raio).

A sua tarefa é descrever duas classes uma representando um **ponto** e a outra um circunferência, e implementar um método na classe que representa o circunferência que decida se um ponto está **dentro da circunferência**. Note que a classe **Circunferencia** terá como atributo um objeto da classe **Ponto** para armazenar o seu centro.

Escreva também uma classe cliente para testar sua implementação.

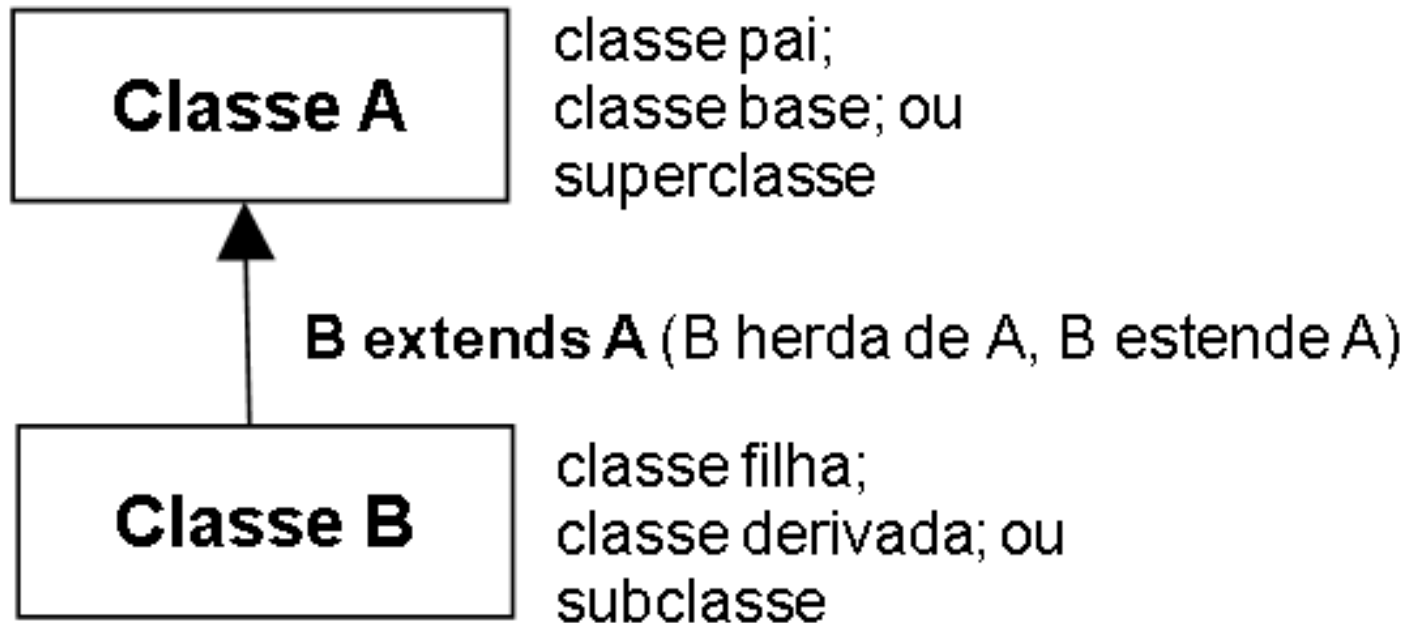
Relacionamento semântico: herança

- O conceito de **herança** é uma das palavras-chave que caracterizam o paradigma de programação orientada a objetos (POO), juntamente com os mecanismos de abstração de tipos abstratos de dados e polimorfismo.
- **Herança** é um tipo especial de **relacionamento do tipo generalização-especialização**, onde uma **subclasse** (mais **especializada**) herda parte (ou o todo) de uma **superclasse** (mais **genérica**).



Relacionamento semântico: herança

- No diagrama abaixo, a **subclasse B** (mais especializada) herda parte (ou o todo) da **superclasse A** (mais genérica).



Relacionamento semântico: herança

- Um dos objetivos principais da herança é fazer **reuso** de um código já implementado nas superclasses e adicionar os detalhes necessários para que a subclasse que esteja estendendo a superclasse seja mais especializada. Assim uma classe derivada pode:
 - herdar os atributos e métodos da classe base;
 - redefinir os métodos herdados.
 - definir novos atributos e métodos;
- Em Java, toda classe que não estende especificamente uma outra classe, é uma **subclasse** da classe **Object**.

Exercício relacionamento semântico: herança

A **classe Object** tem alguns métodos que fazem sentido para todos os objetos, como os métodos **toString()** e **equals()**. Considere a classe **Ponto** abaixo, já com método **toString()** reescrito. Para tornar nossa classe **Ponto** mais funcional poderíamos reescrever método **public boolean equals(Object obj)** da classe **Object**, que verifica se dois objetos são iguais, tente fazer essa alteração e implemente também uma classe cliente para testar suas classes.

```
public class Ponto {  
    private int x, y;  
    public Ponto( int x, int y )  
    { this.x = x;  this.y = y; }  
    @Override  
    public String toString(){  
        return "("+this.x+", "+this.y+")";  
    }  
}
```


Exercício relacionamento semântico: herança

Imagine agora que precisamos trabalhar com um ponto em sistema de coordenadas de 3 dimensões, ou seja, um ponto 3D, a melhor forma de modelar isso é criar uma nova classe **Ponto3D**, que herda as características da classe **Ponto**.

Para que a classe **Ponto3D** (**subclasse**) seja uma extensão da classe **Ponto** (**superclasse**) utilizamos a palavra-chave **extends** seguida pelo nome da **superclasse**.

```
class Ponto3D extends Ponto{
    private int z;
    public Ponto3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
}
```

Exercício relacionamento semântico: herança

No exemplo, a classe **Ponto3D** preserva da **superclasse** os atributos (no caso, os valores das coordenadas *x* e *y*) e o método **equals()** já herdado da classe **Object**.

No construtor da classe **Ponto3D**, estamos reutilizando o construtor da classe **Ponto**, acessado com o apontador **super**. Podemos utilizar de três maneiras:

<code>super(...)</code>	invoca o construtor da superclasse
<code>super.id</code>	acessa o atributo da superclasse identificado por id
<code>super.m(...)</code>	acessa o método da superclasse identificado por m

Quando não houver confusão entre o nome de identificadores dos métodos da superclasse para subclasse, o operador **super** pode ser suprimido

Exercício relacionamento semântico: herança

1) Para tornar nossa classe **Ponto3D** mais funcional poderíamos reescrever método **public boolean equals(Object obj)** da classe **Ponto**, que verifica se dois objetos **Ponto3D** são iguais, tente fazer essa alteração.

2) reescreva o método **distancia()**, implementado na classe **Ponto**, que calcula distância no espaço 2D para agora calcular a distância no espaço 3D.

Para você lembrar como se calcula a distância no espaço 3D acesse:

<https://mundoeducacao.bol.uol.com.br/matematica/distancia-entre-dois-pontos-no-espaco.htm>

Escreva também uma classe cliente para testar sua implementação, a classe cliente deve fazer chamada aos dois métodos reescritos (**equals**, **distancia**).

Relacionamentos com Multiplicidade 1:N

- Relacionamentos com multiplicidade 1:N, entre uma classe A e uma classe B, são interpretados da seguinte forma: para cada instância da classe A, teremos N instâncias da classe B.
- Normalmente, relacionamento 1:N são mapeados da seguinte forma:
 - Criarmos um ***container*** (uma coleção=um vetor, por exemplo) **de objetos B** como um atributo da **classe A**.
 - Aplicamos regras específicas para inserção de cada instância de **B** no ***container***.
- Lembre-se que a capacidade de um vetor é fixa e não é possível redimensionar um vetor em Java, teremos de contornar isso mais adiante.

Relacionamentos com Multiplicidade 1:N

```
public class A{

    private B container[]; // vetor como container
    private int qtd;
    public A( int N ){
        // vetor alocado com N posições
        this.container = new B[N];
        this.qtd = 0;
    }
    public adiciona( B b ){
        //como fazer isso ?
    }
}
```

Exercício relacionamento 1:N

Suponha que queiramos criar uma classe **ListaPonto** para armazenar todos os pontos de um sistema de sistema de computação. Para armazenar os pontos utilizaremos um vetor como *container*.

```
// classe A
public class listaPonto{
    private Ponto pontos[]; // Pontos da classe B
    public listaPonto( int N ){
        // Lista criada com N pontos
        pontos =new Ponto[N];
    }
}
```

Exercício relacionamento 1:N

Agora implemente os métodos para cada uma das operações descritas abaixo:

1) Adicionar um ponto ao final do *container*

```
public void adiciona(Ponto p){....}
```

2) Adicionar um ponto em uma dada posição

```
public void adiciona(Ponto p, int i){....}
```

3) Verificar se dado um ponto está contido no container

```
public int busca(Ponto p){....}
```

4) Remover um ponto em uma dada posição;

```
public Ponto remove(int i){....}
```

5) Informa a quantidade de pontos armazenados no container.

```
public int quantidade(){....}
```

Exercício relacionamento 1:N

Na implementação dos métodos considere que os pontos são armazenados de forma contínua e estável no ***container***, ou seja, não podemos ter “buracos” no meio do vetor e as operações de adicionar e de remover não alteram a posição relativa dos elementos no vetor. Para cada testar suas operações implemente um cliente.

1) adicionar um ponto ao final do container é necessário primeiro encontrar a última posição do *container* ou a primeira posição vazia no vetor, para isso basta percorrer o vetor da esquerda para a direita até achar um valor **null** (lembrando que os vetores em Java guardam referências, e o valor padrão para referências é **null**). Achado a primeira posição vazia é só guardar o ponto nela. Teria como melhor isso ?

Exercício relacionamento 1:N

2) adicionar um ponto em uma dada posição, primeiro é preciso verificar se a posição faz sentido ou não, ou seja, só podemos adicionar um ponto em alguma posição que já estava ocupada ou na primeira posição disponível no final do vetor de pontos. Caso a posição seja válida, devemos tomar cuidado para não colocar um elemento sobre outro. É preciso deslocar todos os elementos a “direita” da posição onde vamos inserir uma vez para a “frente”. Isso abrirá um espaço para guardar o novo elemento.

3) verificar se dado um ponto está contido no container, nessa operação será informado para a método um ponto e o método devolve o índice onde o ponto foi encontrado. Caso o ponto não esteja no container o método devolve -1.

Exercício relacionamento 1:N

4) remover um ponto em uma dada posição, também é preciso verificar se a posição está ocupada ou não, se a posição estiver ocupada então podemos remover a pontuação. Para isso basta deslocar os elementos que estavam a direita daquele que removemos uma vez para esquerda e fechamos o “buraco” aberto pela remoção.

5) A operação que informa a **quantidade de pontos** armazenados na lista de pontos é bem simples e dispensa comentários.

Leitura importante:

No Livro:

Horstmann, C. **Conceitos de Computação com Java**. 5.ed.
Porto Alegre: Bookman, 2009.

Leiam os capítulos: capítulo 10 (Herança) e capítulo 12
Seção 2.3 (Relacionamentos entre Classes)

Na apostila on-line

CAELUM. **Java e orientação a objetos**.

<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/>

Leiam o capítulo 9: Herança, reescrita e polimorfismo

Bons estudos !