Programação Orientada a Objetos (PO24CP)

Aula #09 - Herança

Profª Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Herança - conceitos básicos

Genética

Um organismo adquire características semelhantes à do organismo que o gerou

Programação Orientada a Objetos

Uma classe herda **atributos** e **métodos** de uma **outra classe**

Para que serve em POO?

O conceito de **herança** torna mais rápido o desenvolvimento de softwares complexos

- Novas classes são criadas baseadas em classes existentes
- Objetivo: **reutilização de código**

Herança - conceitos básicos

Genética

Um organismo adquire características semelhantes à do organismo que o gerou

Programação Orientada a Objetos

Uma classe herda **atributos** e **métodos** de uma **outra classe**

Para que serve em POO?

O conceito de **herança** torna mais rápido o desenvolvimento de softwares complexos

- Novas classes são criadas baseadas em classes existentes
- Objetivo: **reutilização de código**

Herança - conceitos básicos

Genética

Um organismo adquire características semelhantes à do organismo que o gerou

Programação Orientada a Objetos

Uma classe herda **atributos** e **métodos** de uma **outra classe**

Para que serve em POO?

O conceito de **herança** torna mais rápido o desenvolvimento de softwares complexos

- Novas classes são criadas baseadas em classes existentes
- Objetivo: **reutilização de código**

Herança - conceitos básicos

- Palavra chave utilizada: **extends**
- Forma geral:
  `class` *subclasse* `extends` *superclasse*...

classe filha, subclasse ou **classe derivada**

- A classe que herda os atributos e funções de outra classe

classe pai, superclasse ou **classe base**

- A classe cujo membros são herdados por outras classes

Quando usar herança?

- Para criar relações *é um* ou *é-um-tipo-de* entre classes
- Ideal para casos onde são necessárias classes distintas para atacar problemas específicos. Porém, tais classes necessitam compartilhar um **núcleo comum**

Herança - conceitos básicos

- Palavra chave utilizada: **extends**
- Forma geral:
  `class` *subclasse* `extends` *superclasse*...

classe filha, subclasse ou **classe derivada**

- A classe que herda os atributos e funções de outra classe

classe pai, superclasse ou **classe base**

- A classe cujo membros são herdados por outras classes

Quando usar herança?

- Para criar relações *é um* ou *é-um-tipo-de* entre classes
- Ideal para casos onde são necessárias classes distintas para atacar problemas específicos. Porém, tais classes necessitam compartilhar um **núcleo comum**

Herança - conceitos básicos

- Palavra chave utilizada: **extends**
- Forma geral:
  `class` *subclasse* `extends` *superclasse*...

classe filha, subclasse ou **classe derivada**

- A classe que herda os atributos e funções de outra classe

classe pai, superclasse ou **classe base**

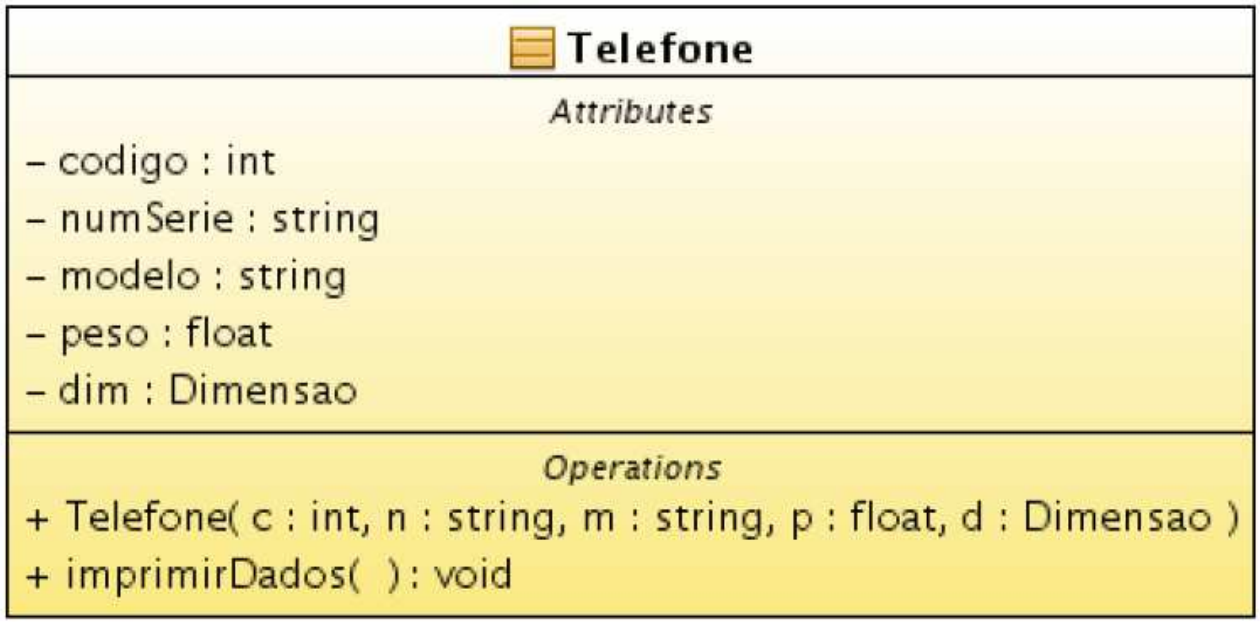
- A classe cujo membros são herdados por outras classes

Quando usar herança?

- Para criar relações *é um* ou *é-um-tipo-de* entre classes
- Ideal para casos onde são necessárias classes distintas para atacar problemas específicos. Porém, tais classes necessitam compartilhar um **núcleo comum**

Exemplo 1: Sistema para cadastro de produtos

- Uma indústria da área de telecomunicações necessita de um sistema para cadastrar os produtos que fabrica
  - **Aparelho telefônico**
- As informações necessárias para o cadastro são:
  - código, número de série, modelo, cor, peso, dimensões (AxLxP)



Exemplo 1: Sistema para cadastro de produtos

- A empresa começou a fabricar também **telefones sem fio**
- Os **telefones sem fio** compartilham todas as características de um **telefone**, porém possuem novas características
  - frequência, quantidade de canais, distância de operação
- O atual sistema não permite cadastrar essas novas informações

O que fazer?

1. Criar uma nova classe `telefone sem fio` e colocar nela tudo o que tem na classe `telefone` mais as características do telefone sem fio?
2. Herdar as características comuns da classe `telefone` e adicionar as particulares do telefone sem fio?

Exemplo 1: Sistema para cadastro de produtos

- A empresa começou a fabricar também **telefones sem fio**
- Os **telefones sem fio** compartilham todas as características de um **telefone**, porém possuem novas características
  - frequência, quantidade de canais, distância de operação
- O atual sistema não permite cadastrar essas novas informações

O que fazer?

1. Criar uma nova classe `telefone sem fio` e colocar nela tudo o que tem na classe `telefone` mais as características do telefone sem fio?
2. Herdar as características comuns da classe `telefone` e adicionar as particulares do telefone sem fio?

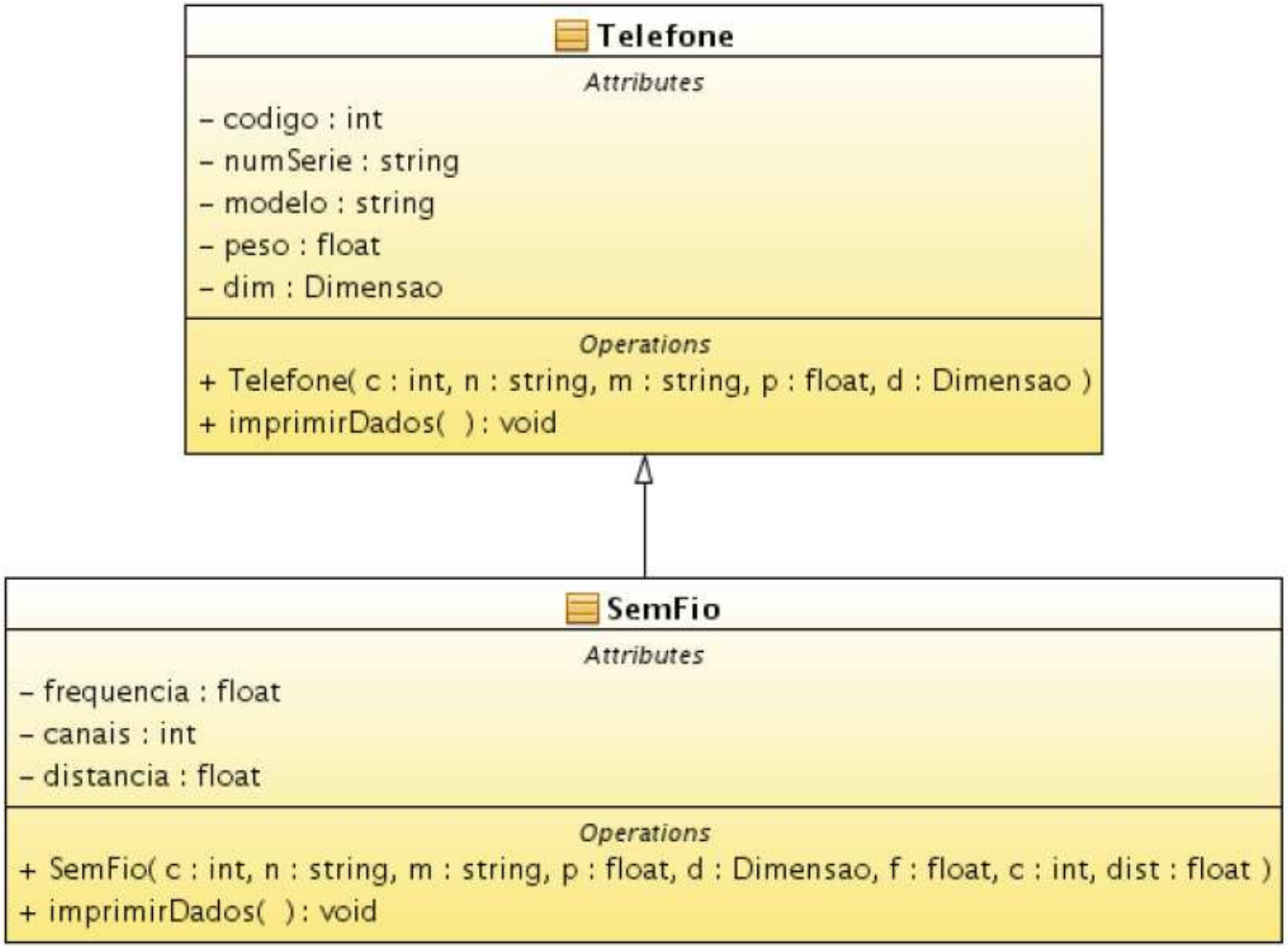
Exemplo 1: Sistema para cadastro de produtos

- A empresa começou a fabricar também **telefones sem fio**
- Os **telefones sem fio** compartilham todas as características de um **telefone**, porém possuem novas características
  - frequência, quantidade de canais, distância de operação
- O atual sistema não permite cadastrar essas novas informações

O que fazer?

1. Criar uma nova classe `telefone sem fio` e colocar nela tudo o que tem na classe `telefone` mais as características do telefone sem fio?
2. Herdar as características comuns da classe `telefone` e adicionar as particulares do telefone sem fio?

Exemplo 1: Sistema para cadastro de produtos - com herança



Superclasse Telefone

```
public class Telefone{
 private int codigo;
 private String numSerie, modelo;
 private float peso;
 private Dimensao dim;

 public Telefone(int c, String s, String m, float p, Dimensao d)
 {
  this.codigo = c; this.peso = p; this.dim = d;
  this.numSerie = s; this.modelo =m;
 }

 public void imprimirDados(){
  System.out.println ("Codigo: " + this.codigo);
    ...
  this.dim.imprimirDados();
 }
}
```

Subclasse SemFio

```
public class SemFio extends Telefone{
 private float frequencia,distancia;
 private int canais;

 public SemFio(int c, String s, String m, float p, Dimensao d,
               int ca, float f, float dis){
    super(c, s, m, p, d); // invocando o construtor da superclasse
    this.frequencia = f;
    this.distancia = dis;
    this.canais = ca;
 }

 // sobreescrita do metodo da superclasse
 public void imprimirDados(){
    super.imprimirDados(); // invocando o metodo de mesmo nome
                           //da superclasse
    System.out.println("Freq: " + this.frequencia);
    ...
 }
}
```

Criando instâncias do Telefone e SemFio

```
public class Principal{
 public static void main(String[] args){

    Telefone t = new Telefone(1,"ABC123","MesaTel",0.5, new
                        Dimensao(10,10,5));

    SemFio sf = new SemFio(2,"DEF456","LivreTel",0.7,new Dimensao
                    (20,8,7), 11, 2400,100);

    t.imprimirDados();
    sf.imprimirDados();
 }
}
```

Sobrescrita de método

Uma subclasse pode sobrescrever um método da superclasse que tenha a **mesma assinatura** (tipo de retorno, nome e lista de parâmetros)

```
public class Telefone{
 public void ola(){
  System.out.println("Ola, sou um telefone"); } }
```

```
public class Semfio extends Telefone{
 public void ola(){
  System.out.println("Ola, sou um telefone sem fio"); } }
```

```
public class Principal{
 public static void main(String args[]){
  Telefone t = new Telefone();
  Semfio s = new Semfio();
  t.ola(); // Ola, sou um telefone
  s.ola(); // Ola, sou um telefone sem fio  }
}
```

Propriedades de Herança

- Uma subclasse não pode acessar os membros privados de sua superclasse.
- Cada classe pode ter no máximo uma superclasse, mas cada superclasse pode ter muitas subclasses.
- Um construtor de subclasse pode chamar um construtor de superclasse por uso de **super()**, **antes de fazer qualquer outra coisa**.
- **Se você não chamar um construtor de superclasse**, o construtor sem argumentos será chamado automaticamente.

Exemplo 2 - Construtores de Subclasse

```
class OneDimPoint {
  int x;
  OneDimPoint(){ x = 3;}
  int getX(){ return x;  }
}

class TwoDimPoint extends OneDimPoint {
  int y;
  TwoDimPoint() { y = 4; } // chama primeiro
                           // OneDimPoint()
                           // automaticamente
  int getY() { return y; }
}
```

Exemplo 3 - Construtores de Subclasse

```
class OneDimPoint {
  int x;
  OneDimPoint(int startX) { x = startX; }
  int getX() { return x; }
}

class TwoDimPoint extends OneDimPoint {
  int y;
  TwoDimPoint(int startX, int startY) {
    super(startX); //chamada explícita do construtor
    y = startY;
  }
  int getY() { return y; }
}
```

Usando **super** para acessar membros

- Você pode usar **super** similar ao **this** para acessar membros da superclasse a partir da subclasse.
- Exemplo:

```
class OneDimPoint {
    int x = 3;
  }

class TwoDimPoint extends OneDimPoint {
    int x = 4;
    int getSum() { return this.x + super.x; }
  }
```

Membros públicos, privados e protegidos

Modificador `private`

Os membros **privados** de uma classe só podem ser acessados pelos demais membros desta mesma classe

Modificador `public`

Os membros **públicos** de uma classe podem ser acessados por qualquer outra classe

Modificador protected

O modificador de acesso `protected` apresenta uma restrição intermediária entre o `private` e o `public`

- Membros **protegidos** podem ser acessados pelos demais membros da classe, pelas demais classes do pacote e pelas **classes derivadas**

Modificador de acesso protected: exemplo

```
package produtos; //Classe Telefone na pasta produtos

public class Telefone{
    private String marca;
    protected String modelo;
    public float peso;
}
```

```
package produtos; //Classe SemFio na pasta produtos

public class SemFio extends Telefone{
   private float frequencia;

   public void modificador(){
      this.frequencia = 900; // acesso ok
      this.modelo = "ABC"; // acesso ok
      this.peso = (float) 0.500000; // acesso ok
      this.marca = "GrandTel"; // erro! Nao permitido
   }
}
```

Modificador de acesso protected: exemplo

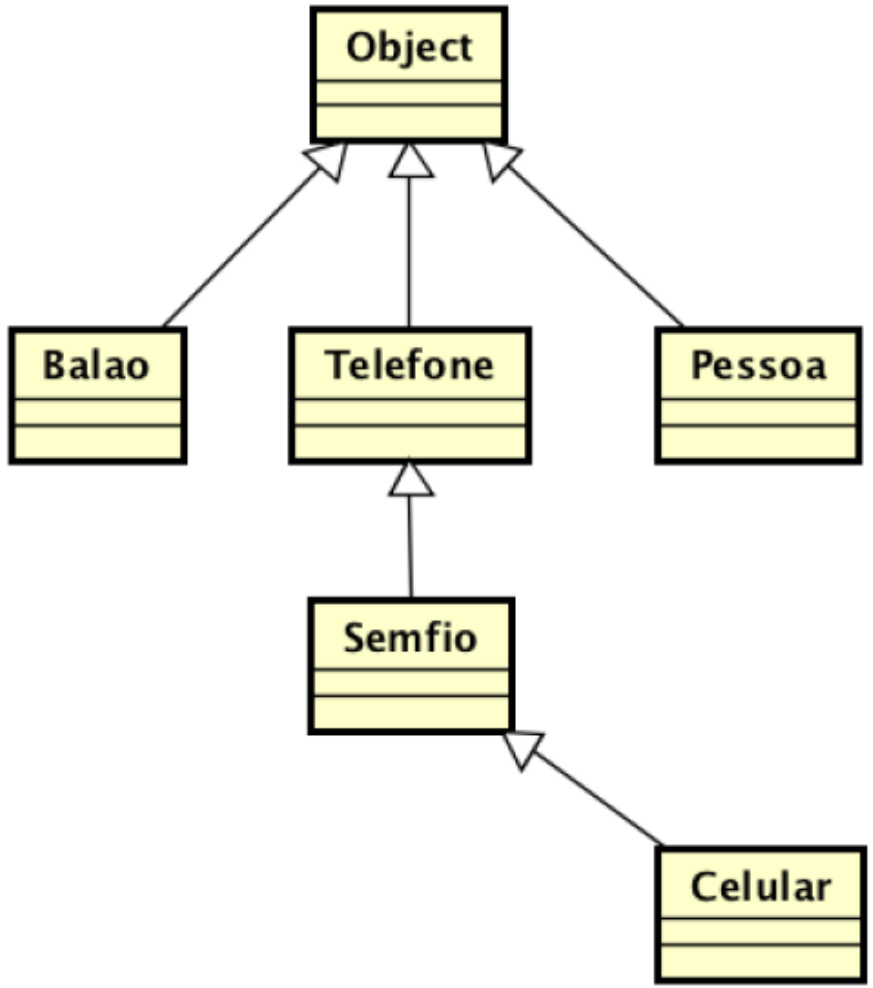
```
package poo;//Classe Principal na pasta poo
import produtos.Telefone;
import produtos.SemFio;

public class Principal{
  public static void main(String[] args){
    Telefone t = new Telefone();
    SemFio sf = new SemFio();

    // invocando um membro public
    t.peso = (float) 0.6; // acesso ok
    // invocando um membro protected
    t.modelo = "DEF"; // erro!
    sf.modelo = "wqa"; // erro!
    sf.modificador(); //através do método correto!
    // invocando um membro private
    t.marca = "GT"; // erro!
  }
}
```

Herança em Java e a classe `Object`

- Com exceção da classe `Object`, que não possui superclasse, toda classe Java tem uma e somente uma superclasse direta
  - Toda classe herda implicitamente da classe `Object`
- Uma classe pode ser derivada de uma outra classe e essa por sua vez pode ser derivada de outra classe, . . .

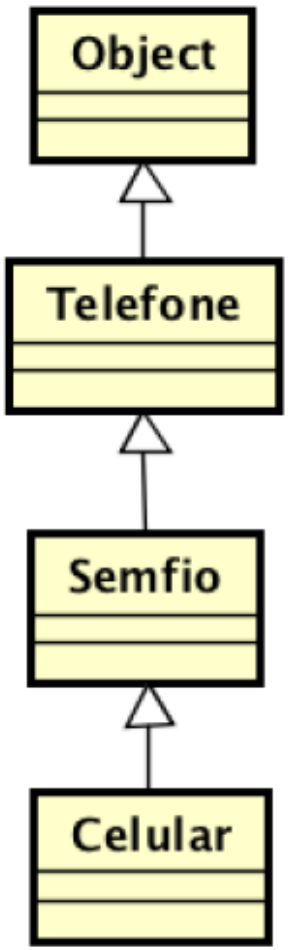


Lembrete

A associação de herança pode ser lida como **é um** ou **é um tipo de**

- Celular **é um** Telefone

Coerção de tipos (*typecasting*) - ou conversão de tipos




```
Telefone a = new Telefone();
Semfio b = new SemFio();
Celular c = new Celular();
```

- Celular **é um** Telefone? SIM!
- Um Telefone pode ser um Celular? Não necessariamente

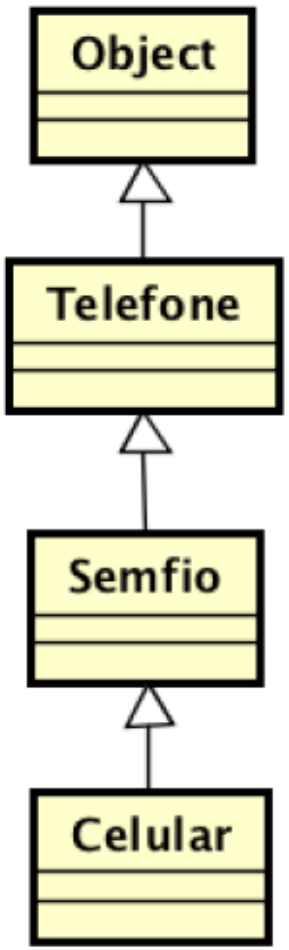
typecasting

O uso do objeto de um tipo na referência de um outro tipo

```
Telefone d = new Celular(); // OK, coerção implícitica
Object e = new Semfio(); // OK, coerção implícitica
Celular f = (Celular) d; //OK, coerção explícitica

Celular g = a; // ERRO! Telefone não é Celular
Celular h = (Celular) e; // ERRO! Semfio não é Celular
```

Coerção de tipos (*typecasting*) - ou conversão de tipos



Operador `instanceof`

Teste lógico para verificar o tipo de um objeto

```
Telefone vetor[] = new Telefone[3];
vetor[0] = new Telefone();
vetor[1] = new Semfio();
vetor[2] = new Celular();

for(int i = 0; i < 3; i++ ){

 if (vetor[i] instanceof Celular){

    Celular c = (Celular) vetor[i];
      ...
 }
}
```

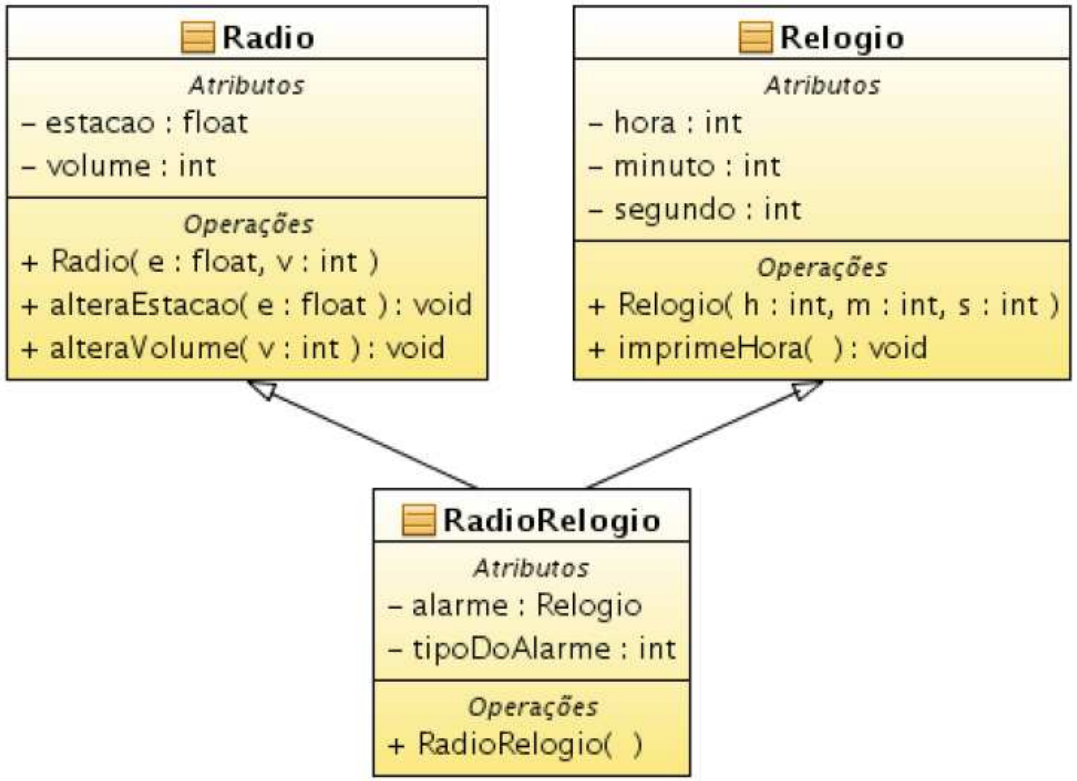
Alguns Métodos da classe `Object`

Todas as classes herdam os métodos da classe `Object`.

Método	**Finalidade**
Object clone()	Cria um novo objeto igual ao objeto
	que está sendo clonado.
boolean equals(Object *objeto*)	Determina se um objeto é igual
	ao outro.
void finalize()	Chamado antes de um objeto não ser
	reciclado.
int hasCode()	Retorna o código hash associado ao
	objeto chamador.
void notify()	Retorna a execução de uma thread que
	está esperando no objeto chamador.
void notifyAll()	Retorna a execução de todas as threads
	que estão esperando no objeto chamador.
String toString()	Retorna uma string que descreve
	o objeto.

Herança Múltipla

- No desenvolvimento de softwares complexos poderemos nos deparar com situações onde uma **nova classe** possui **características semelhantes** com **duas** ou **mais classes** existentes
- A linguagem C++ possui o conceito de **herança múltipla** permitindo que uma classe seja derivada de várias classes base



Herança múltipla em Java

- Em Java uma classe só pode derivar de **uma única classe**.
  - **Não podemos** escrever **X** extends **A**, **B**.
- O conceito de herança múltipla pode ser obtido em Java fazendo uso de **Interfaces**
  - **herança múltipla de tipos** - uma classe pode implementar mais de uma interface
  - **herança múltipla de implementação** - habilidade de herdar as definições de métodos de múltiplas interfaces