

Aula 19 – Herança

Norton Trevisan Roman

28 de maio de 2018

Herança

- Quebramos AreaCasa em Casa, CasaQuad e CasaRet

Herança

- Quebramos AreaCasa em Casa, CasaQuad e CasaRet
- Resta agora modificarmos Residência

```
class Residencia {  
    AreaCasa casa;  
    AreaPiscina piscina;  
  
    Residencia(AreaCasa casa, AreaPiscina  
                piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

Herança

- Quebramos AreaCasa em Casa, CasaQuad e CasaRet
- Resta agora modificarmos Residência
- Como, se queremos poder usar tanto CasaQuad quanto CasaRet?

```
class Residencia {  
    AreaCasa casa;  
    AreaPiscina piscina;  
  
    Residencia(AreaCasa casa, AreaPiscina  
                piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

Herança

- Quebramos AreaCasa em Casa, CasaQuad e CasaRet
- Resta agora modificarmos Residência
- Como, se queremos poder usar tanto CasaQuad quanto CasaRet?
 - Usando a superclasse!

```
class Residencia {  
    AreaCasa casa;  
    AreaPiscina piscina;  
  
    Residencia(AreaCasa casa, AreaPiscina  
                piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

Herança

- Quebramos AreaCasa em Casa, CasaQuad e CasaRet
- Resta agora modificarmos Residência
- Como, se queremos poder usar tanto CasaQuad quanto CasaRet?
 - Usando a superclasse!

```
class Residencia {
    AreaCasa casa;
    AreaPiscina piscina;

    Residencia(AreaCasa casa, AreaPiscina
                piscina) {
        this.casa = casa;
        this.piscina = piscina;
    }
}

class Residencia {
    Casa casa;
    AreaPiscina piscina;

    Residencia(Casa casa, AreaPiscina
                piscina) {
        this.casa = casa;
        this.piscina = piscina;
    }
}
```

Herança

- Podemos agora fornecer tanto um objeto CasaQuad quanto CasaRet a Residência, que funcionará

```
class Residencia {
    AreaCasa casa;
    AreaPiscina piscina;

    Residencia(AreaCasa casa, AreaPiscina
                piscina) {
        this.casa = casa;
        this.piscina = piscina;
    }
}

class Residencia {
    Casa casa;
    AreaPiscina piscina;

    Residencia(Casa casa, AreaPiscina
                piscina) {
        this.casa = casa;
        this.piscina = piscina;
    }
}
```

Herança

- Podemos agora fornecer tanto um objeto CasaQuad quanto CasaRet a Residência, que funcionará
- Subclasses podem ser usadas no lugar da definição da classe. O contrário não, por ser mais específica a subclasse

```
class Residencia {  
    AreaCasa casa;  
    AreaPiscina piscina;  
  
    Residencia(AreaCasa casa, AreaPiscina  
                piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}  
  
class Residencia {  
    Casa casa;  
    AreaPiscina piscina;  
  
    Residencia(Casa casa, AreaPiscina  
                piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```


- Vejamos isso funcionando

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
  
}
```

Herança

- Vejamos isso funcionando

Saída

```
$ java Projeto  
m2 (r1): 1320.0  
m2 (r2): 1523.0
```

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                        r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                        r2.casa.valorM2);  
  
}
```

Herança

- Vejamos isso funcionando

Saída

```
$ java Projeto  
m2 (r1): 1320.0  
m2 (r2): 1523.0
```

- Por que null?

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
  
}
```

Herança

- Vejamos isso funcionando

Saída

```
$ java Projeto  
m2 (r1): 1320.0  
m2 (r2): 1523.0
```

- Por que null?
 - Representa uma casa sem piscina.

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                        r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                        r2.casa.valorM2);  
}
```

- E se fizermos:

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                    null);  
    Residencia r2 = new Residencia(cq,  
                                    null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

Herança

- E se fizermos:

Compilação

```
$ javac Projeto.java
Projeto.java:41: cannot find symbol
symbol   : method area()
location: class Casa
    System.out.println("Área (r1): "
                        +r1.casa.area());
                        ^
1 error
```

```
public static void main(String[] args) {
    CasaRet cr = new CasaRet(10,5,1320);
    CasaQuad cq = new CasaQuad(10,1523);

    Residencia r1 = new Residencia(cr,
                                    null);
    Residencia r2 = new Residencia(cq,
                                    null);

    System.out.println("m2 (r1): "+
                       r1.casa.valorM2);
    System.out.println("m2 (r2): "+
                       r2.casa.valorM2);
    System.out.println("Área (r1): "+
                       r1.casa.area());
}
```

Herança

- E se fizermos:

Compilação

```
$ javac Projeto.java
Projeto.java:41: cannot find symbol
symbol   : method area()
location: class Casa
    System.out.println("Área (r1): "
                        +r1.casa.area());
                        ^
1 error
```

- Por que isso?

```
public static void main(String[] args) {
    CasaRet cr = new CasaRet(10,5,1320);
    CasaQuad cq = new CasaQuad(10,1523);

    Residencia r1 = new Residencia(cr,
                                    null);
    Residencia r2 = new Residencia(cq,
                                    null);

    System.out.println("m2 (r1): "+
                       r1.casa.valorM2);
    System.out.println("m2 (r2): "+
                       r2.casa.valorM2);
    System.out.println("Área (r1): "+
                       r1.casa.area());
}
```

Herança

- Porque o compilador pressupõe que o objeto é Casa

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```


Herança

- Porque o compilador pressupõe que o objeto é Casa
- E Casa não possui um método area

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

Herança

- Porque o compilador pressupõe que o objeto é Casa
- E Casa não possui um método area
- Muito embora ele esteja na memória do objeto

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                    null);  
    Residencia r2 = new Residencia(cq,  
                                    null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

- Quando criamos o objeto, todos os atributos e métodos de sua classe e da superclasse estarão na memória desse objeto

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                        r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                        r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                        r1.casa.area());  
}
```

Herança

- Contudo, só teremos acesso àqueles definidos na declaração da classe invocada

Herança

- Contudo, só teremos acesso àqueles definidos na declaração da classe invocada
- No caso, em Residencia

```
class Residencia {  
    Casa casa;  
    AreaPiscina piscina;  
  
    Residencia(Casa casa,  
                AreaPiscina piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

Herança

- Contudo, só teremos acesso àqueles definidos na declaração da classe invocada
- No caso, em Residencia
- O compilador enxerga somente as definições nas classes e superclasses, não nas subclasses

```
class Residencia {  
    Casa casa;  
    AreaPiscina piscina;  
  
    Residencia(Casa casa,  
                AreaPiscina piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

Herança

- Contudo, só teremos acesso àqueles definidos na declaração da classe invocada
- No caso, em Residencia
- O compilador enxerga somente as definições nas classes e superclasses, não nas subclasses
- Que fazer então?

```
class Residencia {  
    Casa casa;  
    AreaPiscina piscina;  
  
    Residencia(Casa casa,  
                AreaPiscina piscina) {  
        this.casa = casa;  
        this.piscina = piscina;  
    }  
}
```

- Definir um método area em Casa

```
class Casa {  
    double valorM2 = 1500;  
  
    double valor(double area) {  
        if (area >= 0)  
            return(this.valorM2*area);  
        return(-1);  
    }  
  
}
```


- Definir um método `area` em `Casa`

```
class Casa {  
    double valorM2 = 1500;  
  
    double valor(double area) {  
        if (area >= 0)  
            return(this.valorM2*area);  
        return(-1);  
    }  
  
    double area() {  
        return(-1);  
    }  
}
```

Herança

- Definir um método `area` em `Casa`
- Que será sobrescrito pelos `area` existentes nas subclasses

```
class Casa {  
    double valorM2 = 1500;  
  
    double valor(double area) {  
        if (area >= 0)  
            return(this.valorM2*area);  
        return(-1);  
    }  
  
    double area() {  
        return(-1);  
    }  
}
```

- E agora? Irá o código funcionar?

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

- E agora? Irá o código funcionar?

Saída

```
$ java Projeto  
m2 (r1): 1320.0  
m2 (r2): 1523.0  
Área (r1): 150.0
```

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

- Embora enxergue a definição apenas nas classes e superclasses, na hora de rodar, vale o código do objeto na memória

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

- Embora enxergue a definição apenas nas classes e superclasses, na hora de rodar, vale o código do objeto na memória
- Mesmo que este seja subclasse da definição

```
public static void main(String[] args) {  
    CasaRet cr = new CasaRet(10,5,1320);  
    CasaQuad cq = new CasaQuad(10,1523);  
  
    Residencia r1 = new Residencia(cr,  
                                     null);  
    Residencia r2 = new Residencia(cq,  
                                     null);  
  
    System.out.println("m2 (r1): "+  
                       r1.casa.valorM2);  
    System.out.println("m2 (r2): "+  
                       r2.casa.valorM2);  
    System.out.println("Área (r1): "+  
                       r1.casa.area());  
}
```

Polimorfismo

- “O que possui várias formas”

Polimorfismo

- “O que possui várias formas”
- Capacidade de se usar o mesmo nome para métodos diferentes, implementados em diferentes níveis de uma hierarquia de classes

Polimorfismo

- “O que possui várias formas”
- Capacidade de se usar o mesmo nome para métodos diferentes, implementados em diferentes níveis de uma hierarquia de classes
- Para cada classe, tem-se um comportamento específico para o método

Polimorfismo

- Será rodado o código da classe para a qual se tem o objeto

Polimorfismo

- Será rodado o código da classe para a qual se tem o objeto
- Não importa se foi declarada uma superclasse dela

Polimorfismo

- Será rodado o código da classe para a qual se tem o objeto
- Não importa se foi declarada uma superclasse dela
 - Caso em que a superclasse é usada apenas para verificar a assinatura do método, quando da compilação

Polimorfismo

- Sobrecarga (Overloading):

Polimorfismo

- Sobrecarga (Overloading):
 - Métodos com o mesmo nome, porém assinaturas diferentes

Polimorfismo

- Sobrecarga (Overloading):
 - Métodos com o mesmo nome, porém assinaturas diferentes
- Sobrescrita (Overriding):

Polimorfismo

- Sobrecarga (Overloading):
 - Métodos com o mesmo nome, porém assinaturas diferentes
- Sobrescrita (Overriding):
 - Redefinição de um método em classes diferentes, dentro de uma estrutura de herança

Polimorfismo

- Sobrecarga (Overloading):
 - Métodos com o mesmo nome, porém assinaturas diferentes
- Sobrescrita (Overriding):
 - Redefinição de um método em classes diferentes, dentro de uma estrutura de herança
 - Necessitam ter a mesma assinatura

Herança

- Em suma:

Herança

- Em suma:
 - As subclasses podem acrescentar novos métodos:

Herança

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse

Herança

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse
 - Ex: sobrecarregar métodos da superclasse (mesmo nome, nova assinatura)

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse
 - Ex: sobrecarregar métodos da superclasse (mesmo nome, nova assinatura)
 - Podem também redefinir métodos da superclasse com a mesma assinatura

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse
 - Ex: sobrecarregar métodos da superclasse (mesmo nome, nova assinatura)
 - Podem também redefinir métodos da superclasse com a mesma assinatura
- Boa prática:

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse
 - Ex: sobrecarregar métodos da superclasse (mesmo nome, nova assinatura)
 - Podem também redefinir métodos da superclasse com a mesma assinatura
- Boa prática:
 - Primeiro, construir classes para lidar com o caso mais geral

- Em suma:
 - As subclasses podem acrescentar novos métodos:
 - Ex: métodos de acesso aos atributos específicos da subclasse
 - Ex: novas funcionalidades típicas daquela subclasse
 - Ex: sobrecarregar métodos da superclasse (mesmo nome, nova assinatura)
 - Podem também redefinir métodos da superclasse com a mesma assinatura
- Boa prática:
 - Primeiro, construir classes para lidar com o caso mais geral
 - Em seguida, a fim de tratar os casos especiais, definir classes especializadas – herdadas da primeira classe

- Considere as classes CasaQuad e CasaRet:

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaQuad(double lateral, double  
                valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Considere as classes CasaQuad e CasaRet:

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaQuad(double lateral, double  
                valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Onde estão definidos estes atributos?

- Considere as classes CasaQuad e CasaRet:

```
class CasaQuad extends Casa {
    double lateral = 10;

    CasaQuad() {}
    CasaQuad(double valorM2) {
        this.valorM2 = valorM2;
    }
    CasaQuad(double lateral, double
                valorM2) {
        this(valorM2);
        this.lateral = lateral;
    }

    double area() ...
}
```

```
class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;

    CasaRet() {}
    CasaRet(double valorM2) {
        this.valorM2 = valorM2;
    }
    CasaRet(double lateral, double cquarto)
    {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

- Onde estão definidos estes atributos?
 - Na superclasse Casa

Herança

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM = valorM2;  
    }  
    CasaQuad(double lateral, double  
                                   valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Embora correto, fica estranho termos que olhar o atributo em outra classe (arquivo) para sabermos o que fazer

Herança

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM = valorM2;  
    }  
    CasaQuad(double lateral, double  
                                   valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Deveríamos poder passar essa informação à superclasse

Herança

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM = valorM2;  
    }  
    CasaQuad(double lateral, double  
                                   valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Deveríamos poder passar essa informação à superclasse
- Como?

Herança

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        this.valorM = valorM2;  
    }  
    CasaQuad(double lateral, double  
                                   valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
  
    double area() ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
    CasaRet(double lateral, double cquarto)  
    {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- Deveríamos poder passar essa informação à superclasse
- Como? Com super...

- Voltemos então à classe Casa

```
class Casa {  
    double valorM2 = 1500;
```

```
    double valor(double area) {  
        if (area >= 0) return(this.valorM2*  
                                area);  
        return(-1);  
    }
```

```
    double area() {  
        return(-1);  
    }  
}
```

```
class Casa {  
    double valorM2 = 1500;
```

- Voltemos então à classe Casa
- A ela faltam construtores, algo que passe valor a seus parâmetros

```
double valor(double area) {  
    if (area >= 0) return(this.valorM2*  
                           area);  
    return(-1);  
}  
  
double area() {  
    return(-1);  
}  
}
```

- Voltemos então à classe Casa
- A ela faltam construtores, algo que passe valor a seus parâmetros

```
class Casa {  
    double valorM2 = 1500;  
  
    Casa(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
  
    double valor(double area) {  
        if (area >= 0) return(this.valorM2*  
                                area);  
        return(-1);  
    }  
  
    double area() {  
        return(-1);  
    }  
}
```

- E como ficam CasaQuad e CasaRet?

- E como ficam CasaQuad e CasaRet?

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        super(valorM2);  
    }  
    CasaQuad(double lateral, double  
                valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
    ...  
}
```

- E como ficam CasaQuad e CasaRet?

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        super(valorM2);  
    }  
    CasaQuad(double lateral, double  
                                   valorM2) {  
        this(valorM2);  
        this.lateral = lateral;  
    }  
    ...  
}
```

```
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        super(valorM2);  
    }  
    CasaRet(double lateral, double cquarto) {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- E como ficam CasaQuad e CasaRet?

```
class CasaQuad extends Casa {
    double lateral = 10;

    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double
                                   valorM2) {
        this(valorM2);
        this.lateral = lateral;
    }
    ...
}
```

```
class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;

    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

- Vai funcionar?

Super

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        this(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

Super

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        this(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

Linha de Comando

```
$ javac CasaQuad.java
CasaQuad.java:9: cannot find symbol
symbol   : constructor Casa()
location: class Casa
    CasaQuad() {}
            ^
1 error
```

Super

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        this(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

Linha de Comando

```
$ javac CasaQuad.java
CasaQuad.java:9: cannot find symbol
symbol   : constructor Casa()
location: class Casa
    CasaQuad() {}
           ^
1 error
```

Linha de Comando

```
$ javac CasaRet.java
CasaRet.java:13: cannot find symbol
symbol   : constructor Casa()
location: class Casa
    CasaRet() {}
           ^
CasaRet.java:25: cannot find symbol
symbol   : constructor Casa()
location: class Casa
    CasaRet(double lateral, double cquarto) {
           ^
2 errors
```

- O que houve?

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        super(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

- O que houve?
 - Ao definirmos construtores para a superclasse, matamos seu construtor padrão

```
class CasaQuad extends Casa {  
    double lateral = 10;  
    CasaQuad() {}  
    CasaQuad(double valorM2) {  
        super(valorM2);  
    }  
    CasaQuad(double lateral, double valorM2) {  
        super(valorM2);  
        this.lateral = lateral;  
    }  
    ...  
}  
  
class CasaRet extends Casa {  
    double cquarto = 10;  
    double lateral = 10;  
    CasaRet() {}  
    CasaRet(double valorM2) {  
        super(valorM2);  
    }  
    CasaRet(double lateral, double cquarto) {  
        this.lateral = lateral;  
        this.cquarto = cquarto;  
    }  
    ...  
}
```

- O que houve?
 - Ao definirmos construtores para a superclasse, matamos seu construtor padrão
- E quando fizemos a chamada ao construtor padrão da superclasse?

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        super(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

- Quando não chamamos `super` explicitamente nos construtores das subclasses

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        super(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

Herança

- Quando não chamamos `super` explicitamente nos construtores das subclasses
- Nesse caso, o compilador irá inserir uma chamada a `super()`

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        super(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```


Herança

- Quando não chamamos `super` explicitamente nos construtores das subclasses
- Nesse caso, o compilador irá inserir uma chamada a `super()`
- E `Casa()` não existe

```
class CasaQuad extends Casa {
    double lateral = 10;
    CasaQuad() {}
    CasaQuad(double valorM2) {
        super(valorM2);
    }
    CasaQuad(double lateral, double valorM2) {
        super(valorM2);
        this.lateral = lateral;
    }
    ...
}

class CasaRet extends Casa {
    double cquarto = 10;
    double lateral = 10;
    CasaRet() {}
    CasaRet(double valorM2) {
        super(valorM2);
    }
    CasaRet(double lateral, double cquarto) {
        this.lateral = lateral;
        this.cquarto = cquarto;
    }
    ...
}
```

- E como resolvemos esse problema?

- E como resolvemos esse problema?
- inserindo em Casa o construtor padrão

```
class Casa {  
    double valorM2 = 1500;  
  
    Casa(){}  
  
    Casa(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
  
    double valor(double area) {  
        if (area >= 0) return(this.valorM2*  
                                area);  
        return(-1);  
    }  
  
    double area() {  
        return(-1);  
    }  
}
```

- Assim como `this` se refere a esta classe, `super` é referência para a superclasse

Herança

- Assim como `this` se refere a esta classe, `super` é referência para a superclasse
- Assim como com `this()`, qualquer chamada ao construtor `super()` deve ser feita na primeira linha do construtor em que se faz a chamada

- O que acontece se fizermos:

```
class CasaQuad extends Casa {  
    ...  
    double area() {  
        return(super.area());  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.area());  
    }  
}
```

- O que acontece se fizermos:

Linha de Comando

```
$ java CasaQuad  
-1.0
```

```
class CasaQuad extends Casa {  
    ...  
    double area() {  
        return(super.area());  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.area());  
    }  
}
```

Herança

- O que acontece se fizermos:

Linha de Comando

```
$ java CasaQuad  
-1.0
```

```
class CasaQuad extends Casa {  
    ...  
    double area() {  
        return(super.area());  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.area());  
    }  
}
```

- Dentro de uma classe, `super` será uma referência à superclasse

Herança

- O que acontece se fizermos:

Linha de Comando

```
$ java CasaQuad  
-1.0
```

```
class CasaQuad extends Casa {  
    ...  
    double area() {  
        return(super.area());  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.area());  
    }  
}
```

- Dentro de uma classe, `super` será uma referência à superclasse
 - Assim, mesmo que `AreaQuad` sobrescreva `area()`, irá rodar `area()` da superclasse

Então...

Então...

- Sempre haverá uma chamada ao construtor da superclasse, quer explicitemos isso ou não

Então...

- Sempre haverá uma chamada ao construtor da superclasse, quer explicitemos isso ou não
- O construtor da superclasse irá, por sua vez, chamar o construtor da superclasse dela. E assim por diante

Então...

- Sempre haverá uma chamada ao construtor da superclasse, quer explicitemos isso ou não
- O construtor da superclasse irá, por sua vez, chamar o construtor da superclasse dela. E assim por diante
- Mas Casa tem superclasse?

Então...

- Sempre haverá uma chamada ao construtor da superclasse, quer explicitemos isso ou não
- O construtor da superclasse irá, por sua vez, chamar o construtor da superclasse dela. E assim por diante
- Mas Casa tem superclasse?
 - Sim, se não definirmos explicitamente ela será subclasse de `Object`

Então...

- Toda classe em java, que não tenha superclasse, será subclasse de `Object`

Então...

- Toda classe em java, que não tenha superclasse, será subclasse de `Object`
- Temos assim um encadeamento de construtores

Então...

- Toda classe em java, que não tenha superclasse, será subclasse de `Object`
- Temos assim um encadeamento de construtores
 - Ao chamarmos um construtor, todos os das classes acima na hierarquia serão chamados, até se chegar a `Object()`

- O que faz o código em main?

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Herança

- O que faz o código em main?

Linha de Comando

```
$ java CasaQuad  
150000.0  
50000.0
```

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Herança

- O que faz o código em main?

Linha de Comando

```
$ java CasaQuad  
150000.0  
50000.0
```

- Problema de segurança!

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Herança

- Não podemos deixar o valor do m^2 ser mudado a toda hora

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Herança

- Não podemos deixar o valor do m^2 ser mudado a toda hora
- Se criamos uma casa, deve ser com aquele valor passado quando da criação do objeto!

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Herança

- Não podemos deixar o valor do m^2 ser mudado a toda hora
- Se criamos uma casa, deve ser com aquele valor passado quando da criação do objeto!
- Como?

```
class CasaQuad extends Casa {  
    double lateral = 10;  
  
    CasaQuad() {}  
  
    double area() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

- Declarando o atributo como privado em Casa:

```
class Casa {  
    private double valorM2 = 1500;  
  
    Casa(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
  
    double valor(double area) {  
        if (area >= 0) return(this.valorM2*  
                                area);  
        return(-1);  
    }  
    ...  
}
```


- Declarando o atributo como privado em Casa:
- Somente métodos declarados dentro da própria classe poderão enxergar esse atributo

```
class Casa {  
    private double valorM2 = 1500;  
  
    Casa(double valorM2) {  
        this.valorM2 = valorM2;  
    }  
  
    double valor(double area) {  
        if (area >= 0) return(this.valorM2*  
                                area);  
        return(-1);  
    }  
    ...  
}
```

- E os de fora?

```
class CasaQuad extends Casa {  
    ...  
    public static void main(String[] args) {  
        CasaQuad c = new CasaQuad();  
  
        System.out.println(c.valor(c.area()));  
  
        c.valorM2 = 500;  
  
        System.out.println(c.valor(c.area()));  
    }  
}
```

Private

- E os de fora?

Linha de Comando

```
$ javac CasaQuad.java
CasaQuad.java:46: valorM2 has
private access in Casa
    c.valorM2 = 500;
    ^
1 error
```

```
class CasaQuad extends Casa {
    ...
    public static void main(String[] args) {
        CasaQuad c = new CasaQuad();

        System.out.println(c.valor(c.area()));

        c.valorM2 = 500;

        System.out.println(c.valor(c.area()));
    }
}
```

Private

- E os de fora?

Linha de Comando

```
$ javac CasaQuad.java
CasaQuad.java:46: valorM2 has
private access in Casa
    c.valorM2 = 500;
    ^
1 error
```

- Erro de compilação!
(por mais que todos estejam na memória do objeto da subclasse)

```
class CasaQuad extends Casa {
    ...
    public static void main(String[] args) {
        CasaQuad c = new CasaQuad();

        System.out.println(c.valor(c.area()));

        c.valorM2 = 500;

        System.out.println(c.valor(c.area()));
    }
}
```

Não há.