

# Java Herança

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2017/18

1

## Relações entre Classes

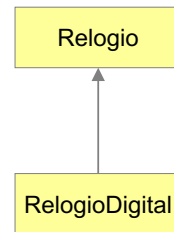
- Parte do processo de modelação em classes consiste em:
  - Identificar entidades candidatas a classes
  - Identificar relações entre estas entidades
- As relações entre classes identificam-se facilmente recorrendo a alguns modelos reais.
  - Por exemplo, um RelógioDigital e um RelógioAnalógico são ambos tipos de Relógio (**especialização** ou **herança**).
  - Um RelógioDigital, por seu lado, contém uma Pilha (**composição**).
- Relações:
  - IS-A
  - HAS-A

2

## Herança (IS-A)

- **IS-A** indica especialização (herança) ou seja, quando uma classe é um sub-tipo de outra classe.
- Por exemplo:
  - Pinheiro é uma (IS-A) Árvore.
  - Um RelógioDigital é um (IS-A) Relógio.

```
class Relogio {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    /* ... */  
}
```

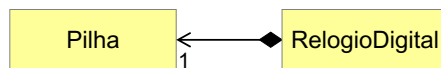


3

## Composição (HAS-A)

- **HAS-A** indica que uma classe é composta por objetos de outra classe.
- Por exemplo:
  - Floresta contém (HAS-A) Árvores.
  - Um RelógioDigital contém (HAS-A) Pilha.

```
class Pilha {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    Pilha p;  
    /* ... */  
}
```



4

## Reutilização de classes

- Sempre que necessitamos de uma classe, podemos:
  - Recorrer a uma classe já existente que cumpre os requisitos
  - Escrever uma nova classe a partir "do zero"
  - Reutilizar uma classe existente usando composição
  - Reutilizar uma classe existente através de herança

5

## Identificação de Herança

- Sinais típicos de que duas classes têm um relacionamento de herança
  - Possuem aspectos comuns (dados, comportamento)
  - Possuem aspectos distintos
  - Uma é uma especialização da outra
- Exemplos:
  - Rato é um Mamífero
  - BTT é uma Bicicleta
  - Cerveja é uma Bebida

6

## Questões?

- Quais as relações entre:
  1. Trabalhador, Motorista, Vendedor, Administrativo e Contabilista
  2. Quadrado, Triângulo, Retângulo, e Losango
  3. Professor, Aluno e Funcionário
  4. Autocarro, Viatura, Roda, Motor, Pneu, Jante

7

## Questões?

- Modelar stock de uma livraria...
  - Livro
  - Artigo
  - Jornal
  - Publicação
  - Autor
  - Periódico
  - Editora
  - LivroEditado
  - Revista

8

## Questões?

- Modelar os *gadgets* de casa...
  - Telemóvel
  - Reprodutor de Áudio
  - Bateria
  - Carregador
  - MP3
  - Auscultador
  - Calculadora

9

## Herança - Conceitos

- A herança é uma das principais características da POO
- A classe CDeriv herda, ou é derivada, de CBase quando CDeriv representa um sub-conjunto de CBase
- A herança representa-se na forma:

```
class CDeriv extends CBase { /* ... */ }
```
- Cderiv tem acesso aos dados e métodos de CBase
  - que não sejam privados em CBase
- Uma classe base pode ter múltiplas classes derivadas mas uma classe derivada não pode ter múltiplas classes base
  - Em Java não é possível a herança múltipla
- Terminologia
  - B é a classe Base / A é derivada de B
  - B é a classe Mãe (Parent) / A é a classe Filha (Child)
  - B é a classe Super / A é a classe Sub

10

## Herança - Exemplo

```
package heranca;
class Person {
    private String name;
    Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON";}
}
class Student extends Person {
    private int nmec;
    Student(String s, int n) { super(s); nmec=n; }
    public int num() { return nmec; }
    public String toString() { return "STUDENT"; }
}
public class Test {
    public static void main(String[] args) {
        Person p = new Person("Joaquim");
        Student stu = new Student("Andreia", 55678);
        System.out.println(p + " : " + p.name());
        System.out.println(stu + " : " + stu.name() + ", " + stu.num());
    }
}
```

Base

Derivada

PERSON : Joaquim  
STUDENT : Andreia, 55678

11

## Herança - Exemplo

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}
class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constr.");
    }
}
public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constr.");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

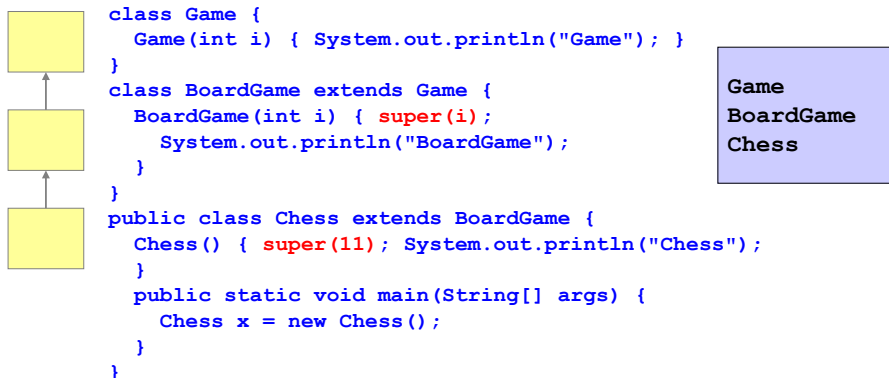
Art constructor  
Drawing constr.  
Cartoon constr.

A construção é feita a partir  
da classe base

12

## Construtores com parâmetros

- Em construtores com parâmetros o construtor da classe base é a primeira instrução a aparecer num construtor da classe derivada.



13

## Herança de Métodos

- Ao herdar métodos podemos:
  - mantê-los inalterados,
  - acrescentar-lhe funcionalidades novas ou
  - redefini-los

14

## Herança de Métodos - herdar

```
class Person {
    private String name;
    Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON";}
}
class Student extends Person {
    private int nmec;
    Student(String s, int n) { super(s); nmec=n; }
    public int num() { return nmec; }
    public String toString() { return "STUDENT"; }
}
public class Test {
    public static void main(String[] args) {
        Student stu = new Student("Andreia", 55678);
        System.out.println(stu + " : " +
            stu.name() + ", " + stu.num());
    }
}
```

15

## Herança de Métodos - redefinir

```
class Person {
    private String name;
    Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON";}
}

class Student extends Person {
    private int nmec;
    Student(String s, int n) { super(s); nmec=n; }
    public int num() { return nmec; }
    public String toString() { return "STUDENT"; }
}
```

16



## Herança de Métodos - estender

```
class Person {
    private String name;
    Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON"; }
}

class Student extends Person {
    private int nmec;
    Student(String s, int n) { super(s); nmec=n; }
    public int num() { return nmec; }
    public String toString()
    { return super.toString() + " STUDENT"; }
}
```

17

## Herança e controlo de acesso

- Não podemos reduzir a visibilidade de métodos herdados numa classe derivada
  - Métodos declarados como *public* na classe base devem ser *public* nas subclasses
  - Métodos declarados como *protected* na classe base devem ser *protected* ou *public* nas subclasses. Não podem ser *private*
  - Métodos declarados sem controlo de acesso (default) não podem ser *private* em subclasses
  - Métodos declarados como *private* não são herdados

18

## Final

- O classificador final indica "não pode ser mudado"
- A sua utilização pode ser feita sobre:
  - Dados - constantes  
`final int i1 = 9;`
  - Métodos - não redefiníveis  
`final int swap(int a, int b) { //:  
}`
  - Classes - não herdadas  
`final class Rato { //...  
}`
- "final" fixa como constantes atributos de tipos primitivos mas não fixa objetos nem arrays
  - nestes casos o que é constante é simplesmente a referência para o objeto

19

```
class Value { int i = 1; }
public class FinalData {
    // Can be compile-time constants
    private final int i1 = 9;
    private static final int VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 39;

    public final int i4 = (int) (Math.random()*20);
    public static final int i5 = (int) (Math.random()*20);

    private Value v1 = new Value();
    private final Value v2 = new Value();
    private final int[] a = { 1, 2, 3, 4, 5, 6 }; // Arrays

    public static void main(String[] args) {
        FinalData fd1 = new FinalData();
        //! fd1.i1++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant!
        fd1.v1 = new Value(); // OK -- not final
        for(int i = 0; i < fd1.a.length; i++)
            fd1.a[i]++; // Object isn't constant!
        //! fd1.v2 = new Value(); // Can't change ref
        //! fd1.a = new int[3];
    }
}
```

20

## Final - Dados

- Os dados final podem ser inicializados dentro do construtor

```
class Dummy { }

class BlankFinal {
    final int i = 0; // Initialized final
    final int j; // Blank final
    final Dummy p; // Blank final reference
    // Blank finals MUST be initialized in the
    constructor:

    BlankFinal() {
        j = 1; // Initialize blank final
        p = new Dummy();
    }

    BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Dummy();
    }
}
```

21

## Final - Argumentos

- A associação de "final" a argumentos de métodos garante que essas referências não serão alteradas dentro do método.

```
class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegal -- g is final
        g.spin();
    }
    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g not final
        g.spin();
    }
    // void f(final int i) { i++; } // Can't change
}
```

22

## Exemplo

```
public final class Ponto {
    private double x;
    private double y;

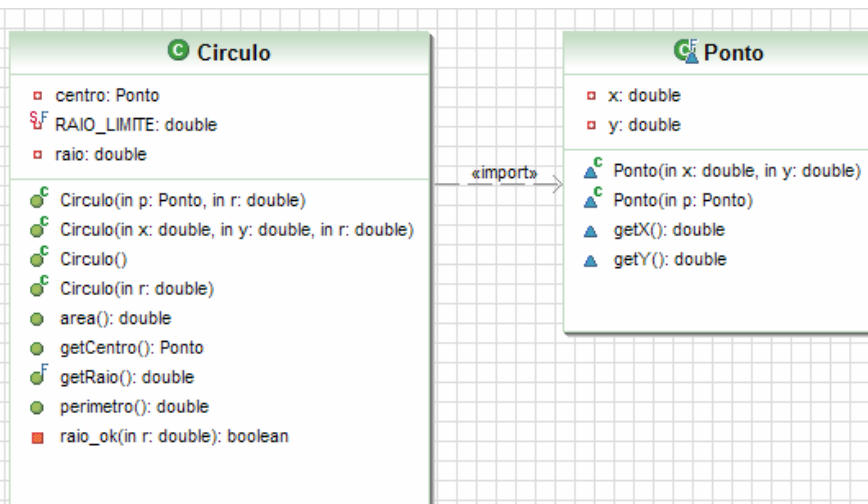
    public Ponto(double x, double y) { this.x=x; this.y=y; }
    public final double x() { return(x); }
    public final double y() { return(y); }
}

public class Circulo {
    private Ponto centro;
    private double raio;

    public static final double RAI0_LIMITE = 100.0;
    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }
    public Circulo(Ponto p, double r) {
        centro = p;
        if (raio_ok(r)) raio = r; else raio = RAI0_LIMITE;
    }
    public Circulo(double x, double y, double r) { this(new Ponto(x, y), r); }
    public double area() { return Math.PI*raio*raio; }
    public double perimetro() { return 2*Math.PI*raio; }
    public final double raio() { return raio; }
    public final Ponto centro() { return centro; }
}
```

23

## Representação UML



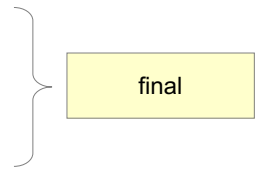
24

## Herança - Boas Práticas

- Programar para a interface e não para a implementação
- Procurar aspectos comuns a várias classes e promovê-los a uma classe base
- Minimizar os relacionamentos entre objetos e organizar as classes relacionadas dentro de um mesmo package
- Usar herança criteriosamente - sempre que possível favorecer a composição

25

## Métodos comuns a todos os objetos

- Todos as classe em Java derivam da super classe `java.lang.Object`
  - Métodos
    - `toString()`
    - `equals()`,
    - `hashCode()`
    - `finalize()`
    - `clone()`
    - `getClass()`
    - `wait()`
    - `notify()`
    - `notifyAll()`
- 

26

## toString()

- Circulo c1 = new Circulo(1.5, 0, 0);
- System.out.println( c1 );

c1.toString() é invocado automaticamente

**Circulo@1afa3**

- O método toString() deve ser sempre redefinido para ter um comportamento de acordo com o objeto

```
public class Circulo {  
    // ....  
    @Override  
    public String toString() {  
        return "Centro : (" + centro.x() + ", " + centro.y() +  
            ") " + " Raio : " + raio;  
    }  
}
```

**Centro : (1.5, 0) Raio : 0**

27

## equals()

- A expressão **c1 == c2** verifica se as referências c1 e c2 apontam para a mesmo objeto
  - Caso c1 e c2 sejam variáveis automáticas a expressão anterior compara valores
- O métodos **equals()** testa se dois objetos são iguais

```
String s1 = "Aveiro";  
String s2 = "Aveiro";  
System.out.println(s1 == s2);           // true (porquê?)  
System.out.println(s1.equals(s2));      // true  
Ponto p1 = new Ponto(1, 1);  
Ponto p2 = new Ponto(1, 1);  
System.out.println(p1 == p2);           // false  
System.out.println(p1.equals(p2));      // false (porquê?)
```
- equals() deve ser redefinido sempre que os objetos dessa classe puderem ser comparados
  - Circulo, Ponto, Complexo ...

28

## Problemas com equals()

- Propriedades da igualdade
  - reflexiva:  $x.equals(x) \rightarrow true$
  - simétrica:  $x.equals(y) \leftrightarrow y.equals(x)$
  - transitiva:  $x.equals(y) \text{ AND } y.equals(z) \rightarrow x.equals(z)$
- Devemos respeitar o contrato '**Object.equals(Object o)**' !!!

```
public class Circulo {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        ...  
    }  
}
```
- Problemas
  - E se 'obj' for null?
  - E se referenciar um objeto diferente de Circulo?

29

## Circulo.equals()

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Circulo other = (Circulo) obj;  
    // verify if the object's attributes are equals  
    if (centro == null) {  
        if (other.centro != null)  
            return false;  
    } else if (!centro.equals(other.centro))  
        return false;  
    if (raio != other.raio)  
        return false;  
    return true;  
}
```

30

## equals() em Herança

```
class BaseClass {
    public BaseClass( int i ) {
        x = i;
    }
    public boolean equals( Object rhs ) {
        if ( rhs == null ) return false;
        if ( getClass() != rhs.getClass() ) return false;
        if ( rhs == this ) return true;
        return x == ( (BaseClass) rhs ).x;
    }
    private int x;
}

class DerivedClass extends BaseClass {
    public DerivedClass( int i, int j ) {
        super( i );
        y = j;
    }
    public boolean equals( Object rhs ) {
        // Não é necessário testar a classe. Feito em base
        return super.equals( rhs ) && y == ( (DerivedClass) rhs ).y;
    }
    private int y;
}
```

31

## hashCode()

- Sempre que o método *equal()* for reescrito, *hashCode* também deve ser
  - Objetos iguais devem retornar códigos de hash iguais
- O objectivo do hash é ajudar a identificar qualquer objeto através de um número inteiro
  - Usado em HashTables

```
// Circulo.hashCode() - Exemplo muito simples !!!
public int hashCode() {
    return raio * centro.x() * centro.y();
}
//..
Circulo c1 = new Circulo(10,15,27);
Circulo c2 = new Circulo(10,15,27);
Circulo c3 = new Circulo(10,15,28);
```

4050
4050
4200

- A construção de uma boa função de hash não é trivial. Para a sua construção recomendam-se outras fontes

32



## Circulo.hashCode()

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = prime + ((centro == null) ? 0 : centro.hashCode());
    long temp = Double.doubleToLongBits(raio);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    // ^ Bitwise exclusive OR
    // >>> Unsigned right shift
    return result;
}
```

33

## Sumário - Porquê herança?

- Muitos objetos reais apresentam esta característica
- Permite criar classes mais simples com funcionalidades mais estanques e melhor definidas
  - Devemos evitar classes com interfaces muito "extensas"
- Permite reutilizar e estender interfaces e código
- Permite tirar partido do polimorfismo

34