

Classes e Herança

UA.DETI.POO

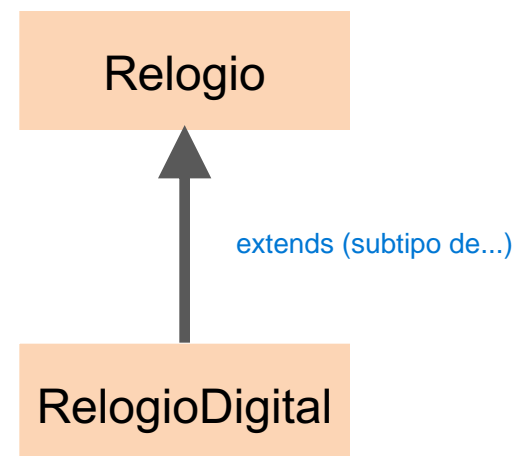
Relações entre Classes

- ❖ 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

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.

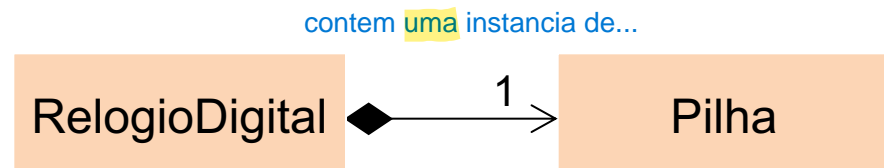
```
class Relogio {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    /* ... */  
}
```



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;  
    /* ... */  
}
```



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**

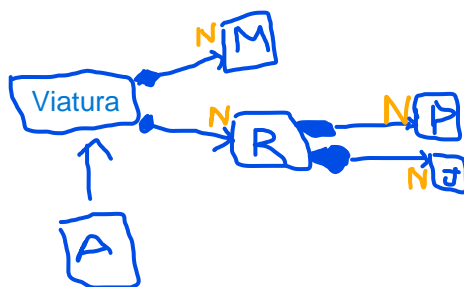
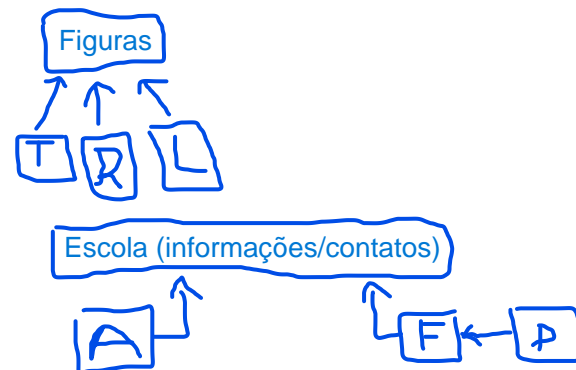
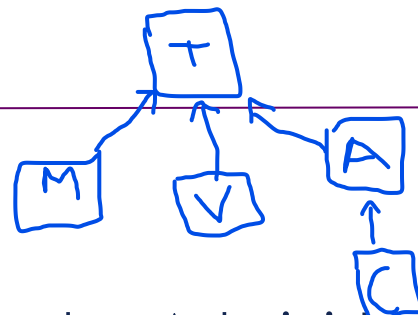
Identificação de Herança

- ❖ Sinais típicos de que duas classes têm um relacionamento de herança
 - Possuem aspetos comuns (dados, comportamento)
 - Possuem aspetos distintos
 - Uma é uma especialização da outra
- ❖ Exemplos:
 - Gato é um Mamífero
 - Circulo é uma Figura
 - Água é uma Bebida

Questões?

❖ Quais as relações entre:

- Trabalhador, Motorista, Vendedor, Aministrativo e Contabilista
- Triângulo, Retângulo e Losango
- Professor, Aluno e Funcionário
- Autocarro, Viatura, Roda, Motor, Pneu, Jante

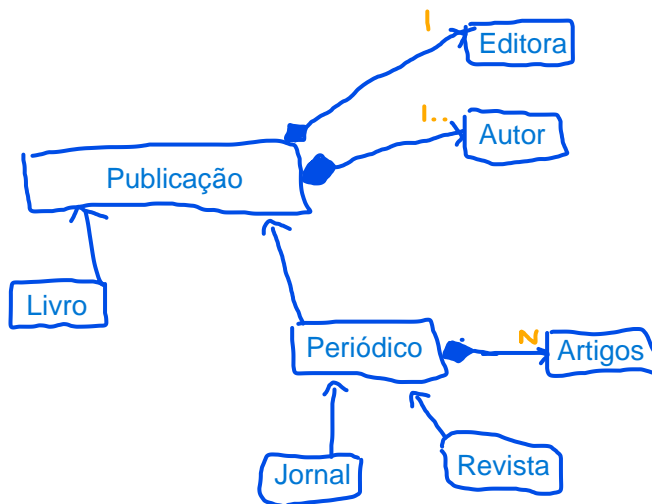


Questões?

Sem plural! Sem acentos e cedilhas

❖ Represente os seguintes elementos (classes) bem como as suas relações (herança e composição)

- Livro
- Artigo
- Jornal
- Publicação
- Autor
- Periódico
- Editora
- LivroEditado
- Revista



Se eu quisesse que a editora também ficasse associada a várias publicações

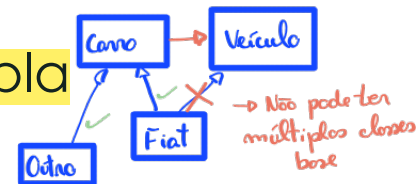
```
class Publicacao(){
    Editora ed;
    Autor[] autores;
    Publicacao(...){
        (...)
    }
}

class Editora(){
    Publicacao[] publicacoes
    Editora(...){
        (...)
    }
}
```


Herança - Conceitos

- ❖ A herança é uma das principais características de 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* (ou "omissão" se estiverem em *pacotes diferentes*)
- ❖ 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



Herança - Exemplo

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

Base

```
class Student extends Person {  
    private int nmec;  
    public Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    @Override do método da class Base → public String toString() { return "STUDENT"; }  
}
```

Derivada

```
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());  
    }  
}
```

PERSON : Joaquim

STUDENT : Andreia, 55678

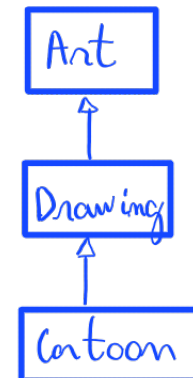
Herança - Exemplo

Sem parâmetros!

```
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



Chamadas implícitas a 'super()' !

Construtores com parâmetros

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

```
class Game {  
    int num;  
    Game(int code) { ... }  
    // ...  
}
```

```
class BoardGame extends Game {  
    // ...  
    BoardGame(int code, int numPlayers) {  
        super(code);  
        // ...  
    }  
}
```

Se não colocar nada, é sempre executado `super()` SEM ARGUMENTOS!
Para enviar parâmetros temos de utilizar `super(...)`

Herda todos os métodos
que não são privados

Herança de Métodos

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

Herança de Métodos - herdar

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

Herança de Métodos - redefinir

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

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

@Override

Herança de Métodos - estender

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

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




estender os métodos do super

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

`public > protected > default > private`

Final

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

```

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];
    }
}

```

Apenas a referência é "final"

<https://tinyurl.com/3j57tb7v>

Exemplo – classe Ponto

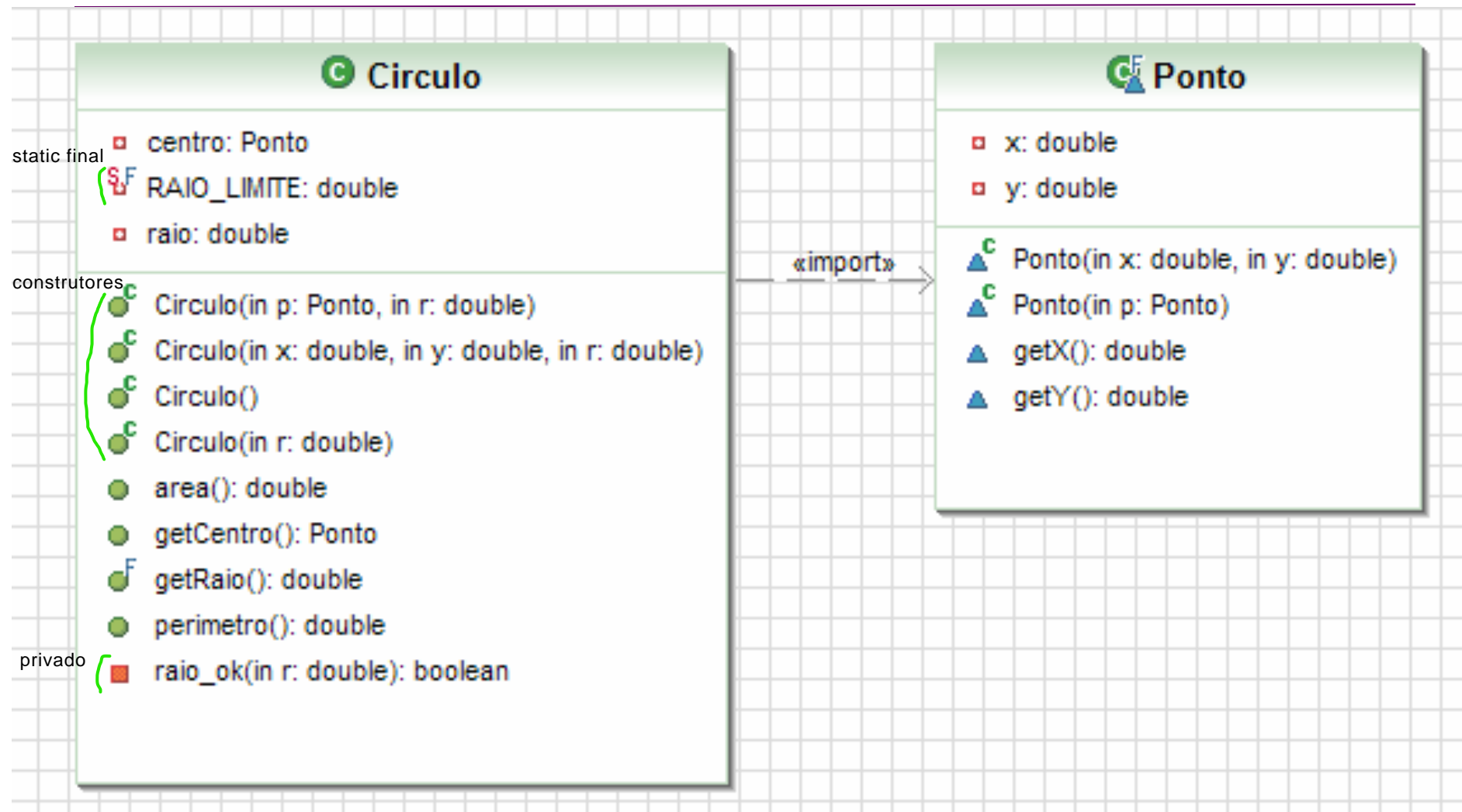
```
public final class Ponto {  
    private double x;  
    private double y;  
  
    public Ponto(double x, double y) { this.x=x; this.y=y; }  
    public final double getX() { return(x); }  
    public final double getY() { return(y); }  
}
```

Exemplo – classe Circulo

```
public class Circulo {  
    private Ponto centro;  
    private double raio;  
  
    public static final double RAIIO_LIMITE = 100.0;  
  
    public Circulo(Ponto p, double r) {  
        centro = p;  
        if (raio_ok(r)) raio = r; else raio = RAIIO_LIMITE;  
    }  
  
    public double area() { return Math.PI*raio*raio; }  
    public double perimetro() { return 2*Math.PI*raio; }  
    public final double getRaio() { return raio; }  
    public final Ponto getCentro() { return centro; }  
  
    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }  
  
}
```

Não podem ser redefinidos em classes derivadas porque são "final"

Representação UML



Herança - Boas Práticas

- ❖ Programar para a interface e não para a implementação
- ❖ Procurar aspetos 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

Métodos comuns a todos os objetos

❖ Em Java, todos as classe derivam da super classe `java.lang.Object`

❖ Métodos desta classe:

– `toString()`

– `equals()`

– `hashCode()`

Deveriam ser redefinidos!

– ~~`finalize()`~~ existem melhores (excluída)

– `clone()` Clone de um objeto ---> `Ponto p1 = new Ponto();`
`p2 = p1.clone();`

– `getClass()`

– `wait()`

– `notify()`

– `notifyAll()`

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    toString() já existe por default para um objeto!  
    public String toString() {  
        return "Centro : (" + centro.x() + ", "  
            + centro.y() + ") " + " Raio : " + raio;  
    }  
}
```

Centro : (1.5, 0) Raio : 0

equals()

- ❖ A expressão `c1 == c2` verifica se as referências `c1` e `c2` apontam para o mesmo objeto

- Caso `c1` e `c2` sejam variáveis automáticas a expressão anterior compara valores

- ❖ O método `equals()` testa se dois objetos são iguais

```
Circulo p1 = new Circulo(0, 0, 1);  
Circulo p2 = new Circulo(0, 0, 1);  
System.out.println(p1 == p2);           // false  
System.out.println(p1.equals(p2));       // false (porquê?)
```

`equals()` do `Object` tem de ser implementado, ele por default compara referências

- ❖ `equals()` deve ser redefinido sempre que os objetos dessa classe puderem ser comparados
- `Circulo`, `Ponto`, `Complexo` ...

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 a assinatura `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?

Circulo.equals()

Forma comum!

@Override

```
public boolean equals(Object obj) {
```

! this é sempre != null

```
    if (this == obj)  referência igual
```

```
        return true;
```

```
    if (obj == null)  objeto passado é "null"
```

```
        return false;
```

```
    if (getClass() != obj.getClass())  são da mesma classe
```

```
        return false;
```

```
    Circulo other = (Circulo) obj;  transformo o objeto na classe do "this"
```

```
        // 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;
```

```
}
```

equals() em Herança

```
class Base {
    private int x;
    public Base ( 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 == ( (Base) rhs ).x;
    }
}

class Derived extends Base {
    public Derived ( 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== ( (Derived) rhs ).y;
    }
    private int y;
}
```

hashCode()

- ❖ Sempre que o método `equal()` for reescrito, `hashCode()` também deve ser

- Objetos iguais devem retornar códigos de hash iguais

- ❖ O objetivo do hash é ajudar a identificar qualquer objeto através de um número inteiro

// 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

Podem existir círculos diferentes com o mesmo hashCode quando a função hash é muito simples

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

Circulo.hashCode()

Função hash mais complexa!

```
@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;
}
```

Igual sempre que tiver os mesmo valores, e é muito POUCO PROVÁVEL que para valores diferentes eles sejam iguais

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 (próxima aula)