

# Programação Orientada por Objetos

---

## **Herança de Classes – Exemplo**

Prof. José Cordeiro,

Prof. Cédric Grueau,

Prof. Laercio Junior

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2019/2020

- Herança – Exemplo Xadrez
  - Redefinição de métodos
- Herança – Exemplo Formas Geométricas
  - Generalização versus especialização de classes
- A Classe Object

# Exemplo — Xadrez

- Requisitos do protótipo:
  - Representar os componentes do jogo sem implementar as regras ou o desenrolar do jogo.
  - Representar as peças: peão, torre, cavalo, rei, rainha e bispo.
  - Representar o tabuleiro de jogo com as posições.
  - Deve ser possível obter em texto a posição de cada peça usando a notação algébrica (ex: e5 – peão na casa e5, ou Te7 – torre na casa e7).



# Exemplo — Xadrez

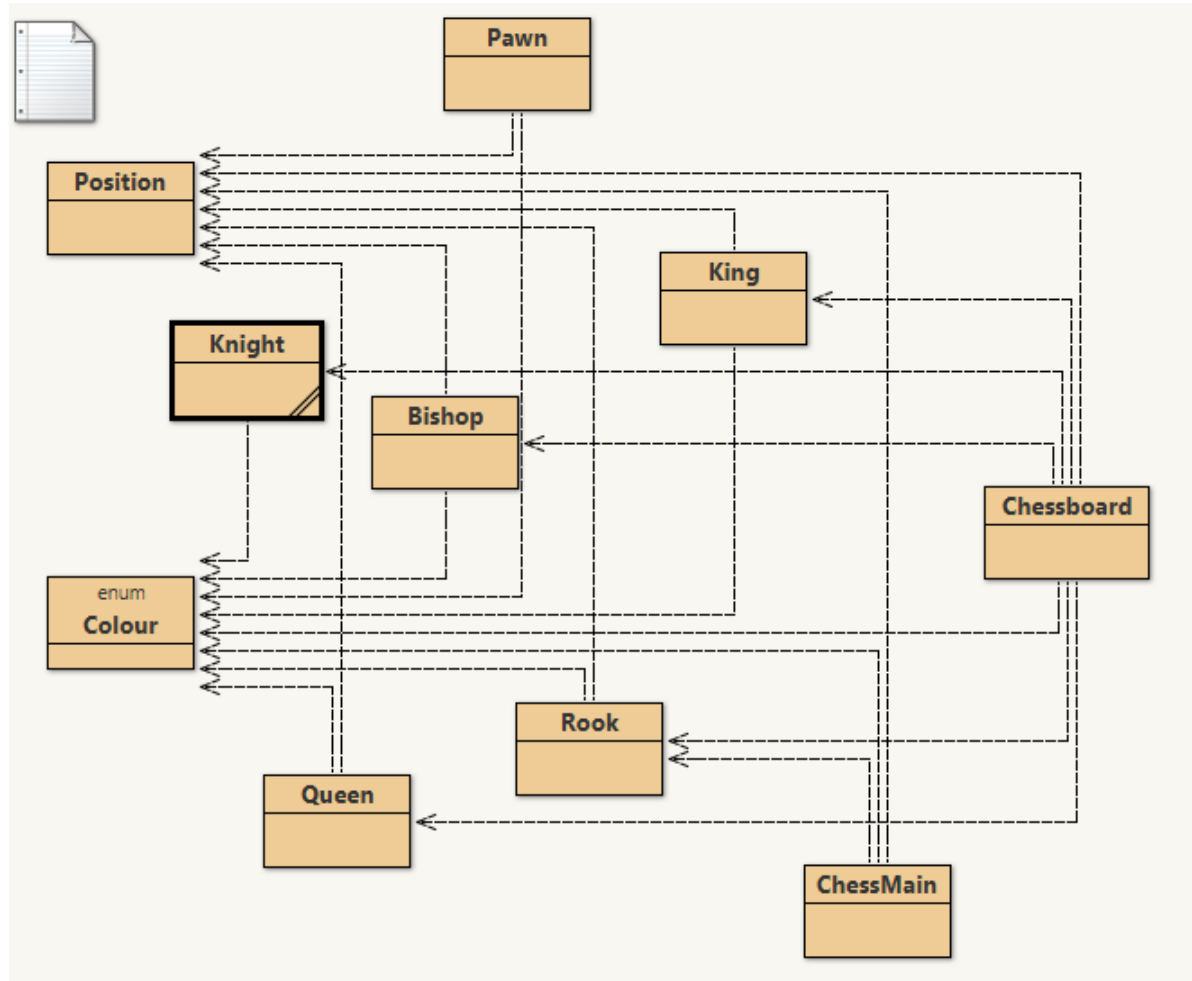
## □ Representação:

- Cada peça poderá ser representada por uma classe
- Peças: peão, torre, cavalo, rei, rainha e bispo.
- O tabuleiro de jogo corresponde a outra classe.



# Exemplo — Xadrez

- Representação 1:
  - Cada peça poderá ser representada por uma classe
  - Peças: peão, torre, cavalo, rei, rainha e bispo.
  - O tabuleiro de jogo corresponde a outra classe.



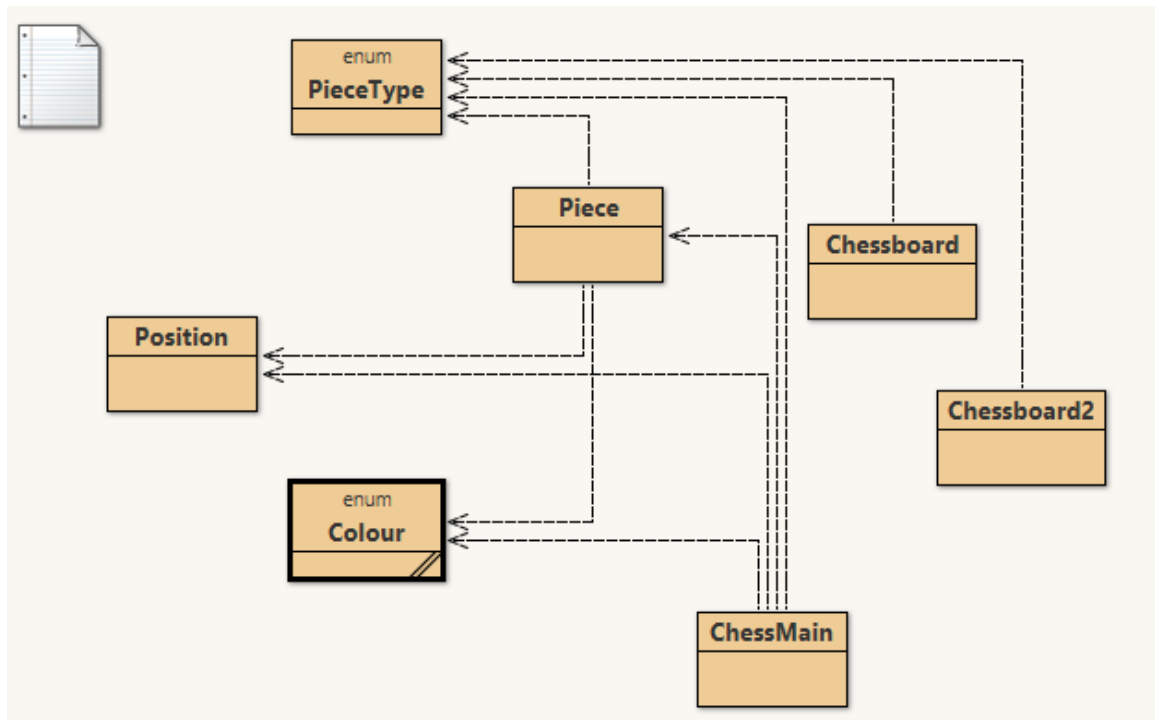
# Exemplo — Xadrez

- Representação 1 – Problemas da solução encontrada:
  - **Duplicação de código** nas classes das peças.
  - A representação do **tabuleiro** ficou **complexa**.
    - Uma lista de peças por tipo de peça (6 listas no total)

# Exemplo — Xadrez

## □ Representação 2:

- Criar uma classe peça que representa qualquer peça
- Usar um atributo **PieceType** que determina qual a peça representada.
- O tabuleiro de jogo corresponde a outra classe.



# Exemplo — Xadrez

- Representação 2 – Problemas da solução encontrada:
  - Classe **Piece** complexa. Tem problemas de coesão.
  - Na classe **Chessboard** ter-se-á de utilizar vários **switch** sempre que se quiser escolher entre os vários tipos de peça.
    - Exemplo: na movimentação das peças.



## Exemplo — Xadrez (3)

- Representação 3 – Usar a herança:
  - Definir uma classe **Piece** como **superclasse**.
    - Inclui os atributos e métodos que são idênticos em todas as peças.
  - Definir cada uma das **peças** como uma **subclasse** da classe **Piece**
  - Na classe **Chessboard** ter uma única lista de peças tirando partido do **princípio da substituição**.
    - Exemplo: na movimentação das peças.

# Exemplo — Xadrez (3)

## ❑ Exemplo de objetos da representação 1

pawn1 : Pawn

|                           |       |             |
|---------------------------|-------|-------------|
| private Colour colour     | WHITE | Inspecionar |
| private Position position |       | Obter       |

Mostrar campos estáticos

Fechar

position : Position

|                |     |             |
|----------------|-----|-------------|
| private char x | 'a' | Inspecionar |
| private int y  | 1   | Obter       |

Mostrar campos estáticos

Fechar

pawn1: Pawn

- herdado de Object
- Colour getColour()
- String getName()
- Position getPosition()
- char getX()
- int getY()
- void setPosition(char x, int y)
- void setPosition(Position position)
- void setY(int y)
- String toString()



rook1 : Rook

|                           |       |             |
|---------------------------|-------|-------------|
| private Colour colour     | BLACK | Inspecionar |
| private Position position |       | Obter       |

Mostrar campos estáticos

Fechar

position : Position

|                |     |             |
|----------------|-----|-------------|
| private char x | 'a' | Inspecionar |
| private int y  | 1   | Obter       |

Mostrar campos estáticos

Fechar

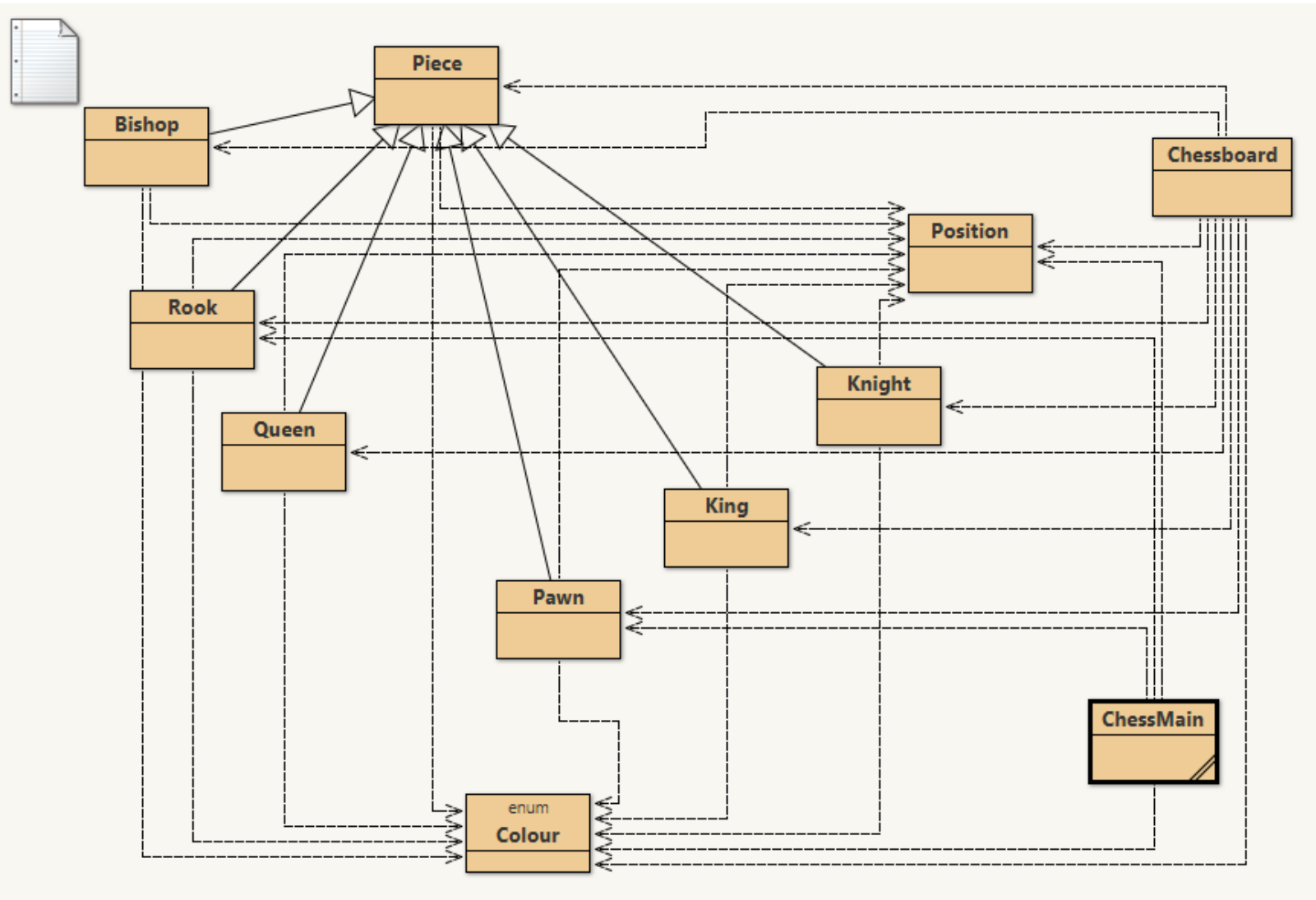


rook1: Rook

- herdado de Object
- Colour getColour()
- String getName()
- Position getPosition()
- char getX()
- int getY()
- void setPosition(char x, int y)
- void setPosition(Position position)
- void setY(int y)
- String toString()

# Exemplo — Xadrez (3)

- Solução com herança de classes:



# Exemplo — Xadrez (3)

## □ Classe **Piece**

```
public class Piece {  
  
    private Colour colour;  
    private Position position;  
  
    public Piece(Colour colour, Position position) {  
        this.colour = colour;  
        if (position != null) {  
            this.position = position;  
        } else {  
            this.position = new Position();  
        }  
    }  
  
    // restante código omitido  
}
```

## Exemplo — Xadrez (3)

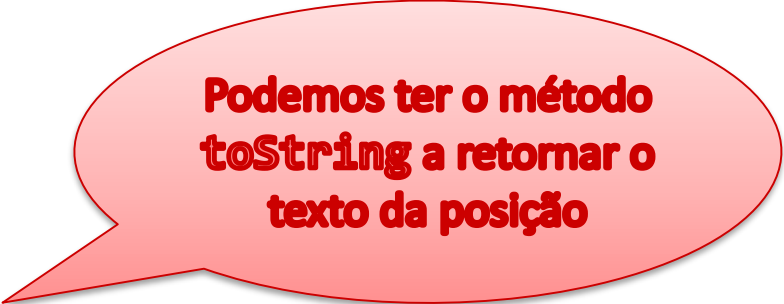
### □ Classe **Piece** – métodos (1/2)

```
public Colour getColour() {  
    return colour;  
}  
  
public Position getPosition() {  
    return new Position(position.getX(), position.getY());  
}  
  
public void setPosition(char x, int y) {  
    position.setX(x);  
    position.setY(y);  
}  
  
public void setPosition(Position position) {  
    position.setX(position.getX());  
    position.setY(position.getY());  
}
```

## Exemplo — Xadrez (3)

### □ Classe **Piece** – métodos (2/2)

```
public void setY(int y) {  
    position.setY(y);  
}  
  
public char getX() {  
    return position.getX();  
}  
  
public int getY() {  
    return position.getY();  
}  
  
@Override  
public String toString() {  
    return position.toString();  
}
```



**Podemos ter o método  
toString a retornar o  
texto da posição**

# Exemplo — Xadrez (3)

## □ Classe **Pawn**

```
public class Pawn extends Piece{  
  
    public Pawn(Colour colour, Position position) {  
        super(colour, position);  
    }  
  
    public String getName() {  
        return "Peão";  
    }  
  
}
```

Deriva da classe **Piece**

Chamada ao **construtor** da classe **Piece** (superclasse)

Este método é diferente em todas as classes das peças

# Exemplo — Xadrez (3)

## □ Classe **Rook**

```
public class Rook extends Piece {
```

Deriva da classe **Piece**

```
    public Rook(Colour colour, Position position) {  
        super(colour, position);  
    }
```

Chamada ao **construtor** da classe **Piece** (superclasse)

```
    public String getName() {  
        return "Torre";  
    }
```

Este método é diferente em todas as classes das peças

```
    @Override  
    public String toString() {  
        return "T" + super.toString();  
    }  
}
```

**O método `toString` que é herdado escreve apenas a posição da peça. É necessário reescrever este método**



# Herança — Redefinição de métodos

- Por vezes os **métodos herdados** da superclasse **não servem** nas subclasses porque estão associados a comportamentos próprios das subclasses.
  - Ex: O método **toString** herdado da classe **Piece** devolve apenas a posição da peça na notação algébrica. Na classe da **Rook** este método deve colocar a letra **'T'** antes da posição
- Neste casos é necessário **redefinir (override)** esse **método**.
  - A palavra **@Override** que aparece em cima do **toString** quer dizer que o método seguinte é a redefinição de um método que já existe.
- No entanto é possível **reutilizar os métodos da superclasse** usando o prefixo **super** seguido de um ponto e do identificador do método que se quer utilizar.

@Override

Indica que se está a **redefinir um método**

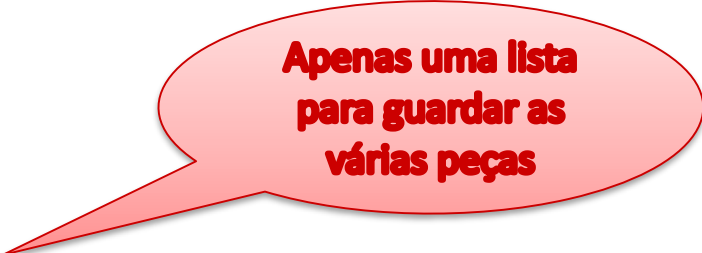
```
public String toString() {  
    return "T" + super.toString();  
}
```

Chamada ao método **toString** da superclasse

## Exemplo — Xadrez (3)

- ❑ Classes **Queen, King, Bishop, Knight**
  - Omitidas: são semelhantes às anteriores
- ❑ Classe **Chessboard**

```
public class Chessboard {  
  
    ArrayList<Piece> pieces;  
  
    public Chessboard() {  
        pieces = new ArrayList<>();  
        setup();  
    }  
  
    // métodos omitidos  
}
```




**Apenas uma lista  
para guardar as  
várias peças**

# Exemplo — Xadrez (3)

## ❑ Classe Chessboard – método **setup**

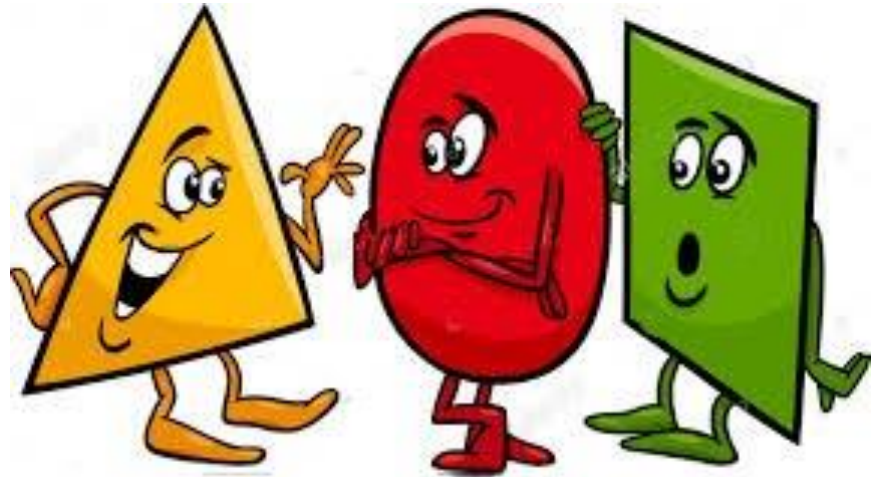
```
private void setup() {  
    for (char x = 'a'; x <= 'h'; x++) {  
        pieces.add(new Pawn(Colour.WHITE, new Position(x, 2)));  
        pieces.add(new Pawn(Colour.BLACK, new Position(x, 7)));  
    }  
    int line = 1;  
    Colour colour = Colour.WHITE;  
    pieces.add(new Rook(colour, new Position('a', line)));  
    pieces.add(new Knight(colour, new Position('b', line)));  
    pieces.add(new Bishop(colour, new Position('c', line)));  
    pieces.add(new Queen(colour, new Position('d', line)));  
    pieces.add(new King(colour, new Position('e', line)));  
    pieces.add(new Bishop(colour, new Position('f', line)));  
    pieces.add(new Knight(colour, new Position('g', line)));  
    pieces.add(new Rook(colour, new Position('h', line)));  
  
    line = 8;  
    colour = Colour.BLACK;  
    pieces.add(new Rook(colour, new Position('a', line)));  
    pieces.add(new Knight(colour, new Position('b', line)));  
    pieces.add(new Bishop(colour, new Position('c', line)));  
    pieces.add(new Queen(colour, new Position('d', line)));  
    pieces.add(new King(colour, new Position('e', line)));  
    pieces.add(new Bishop(colour, new Position('f', line)));  
    pieces.add(new Knight(colour, new Position('g', line)));  
    pieces.add(new Rook(colour, new Position('h', line)));  
}
```



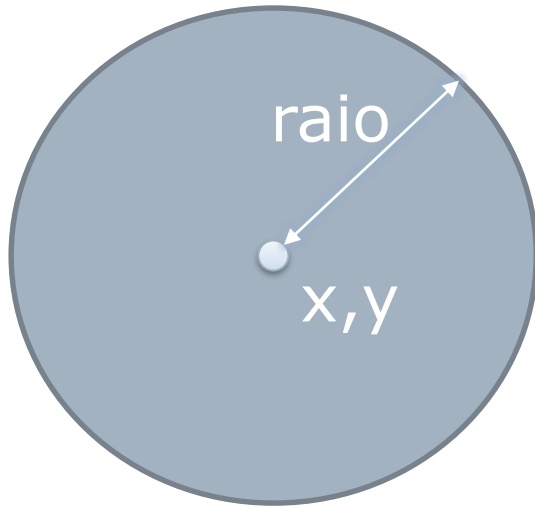
**A inicialização tem  
de ser feita com o  
mesmo detalhe**

# Exemplo — Formas Geométricas

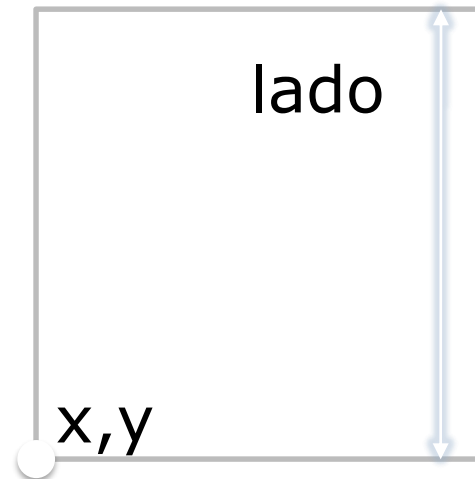
- Requisitos do programa:
  - Desenho de formas geométricas.
  - Representar apenas círculos e quadrados.
    - Deve ser possível saber as dimensões e a posição de cada uma deles.
    - Deve ser possível deslocá-los.



## Exemplo — Formas Geométricas





**Círculo**



**Quadrado**

# Exemplo — Formas Geométricas

|  <b>Circulo</b>   |
|--|
| <i>Attributes</i><br><br>private int x<br>private int y<br>private int raio  |
| <i>Operations</i><br><br>public Circulo( )<br>public Circulo( int x, int y, int raio )<br>public int getX( )<br>public void setX( int x )<br>public int getY( )<br>public void setY( int y )<br>public int getRaio( )<br>public void setRaio( int raio )<br>public void deslocar( int dx, int dy ) |

|  <b>Quadrado</b>  |
|--|
| <i>Attributes</i><br><br>private int x<br>private int y<br>private int lado  |
| <i>Operations</i><br><br>public Quadrado( )<br>public Quadrado( int x, int y, int lado )<br>public int getX( )<br>public void setX( int x )<br>public int getY( )<br>public void setY( int y )<br>public int getLado( )<br>public void setLado( int lado )<br>public void deslocar( int dx, int dy ) |

# Exemplo — Formas Geométricas

```
public class Circulo {  
    private int x, y;  
    private int raio;  
  
    public Circulo() {  
        this.x = 0;  
        this.y = 0;  
        this.raio = 1;  
    }  
  
    public Circulo(int x, int y, int raio) {  
        this.x = x;  
        this.y = y;  
        this.raio = raio;  
    }  
  
    public int getRaio() {  
        return raio;  
    }  
  
    public void setRaio(int raio) {  
        this.raio = raio;  
    }  
}
```

```
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
  
    public void deslocar( int dx, int dy ) {  
        x += dx; y += dy;  
    }  
}
```

# Exemplo — Formas Geométricas

```
public class Quadrado {  
    private int x, y;  
    private int lado;  
  
    public Quadrado() {  
        this.x = 0;  
        this.y = 0;  
        this.lado = 1;  
    }  
  
    public Quadrado(int x, int y, int lado) {  
        this.x = x;  
        this.y = y;  
        this.lado = lado;  
    }  
  
    public int getLado() {  
        return lado;  
    }  
  
    public void setLado(int lado) {  
        this.lado = lado;  
    }  
}
```

```
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
  
    public void deslocar( int dx, int dy ) {  
        x += dx; y += dy;  
    }  
}
```



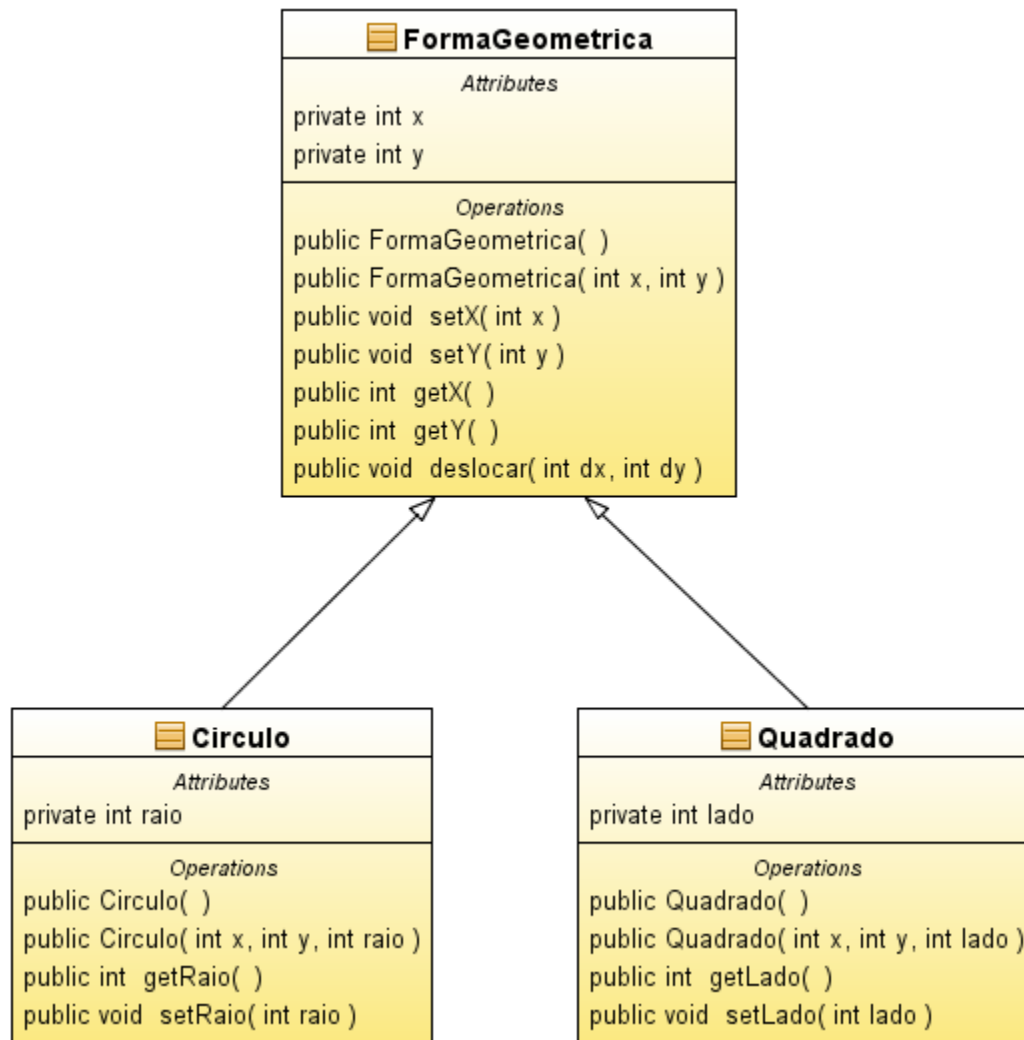
# Exemplo — Formas Geométricas

```
public class Programa {  
  
    public static void main(String[] args) {  
        Circulo circulo = new Circulo(1, 1, 23);  
        Quadrado quadrado = new Quadrado(0, 0, 4);  
  
        System.out.println("Circulo: - Posição (" + circulo.getX() +  
                            "," + circulo.getY() +  
                            ") - Raio: " + circulo.getRaio() );  
  
        System.out.println("Quadrado: - Posição (" + quadrado.getX() +  
                            "," + quadrado.getY() +  
                            ") - Lado: " + quadrado.getLado() );  
  
        quadrado.deslocar( 2, 2);  
  
        System.out.println("Quadrado: - Posição (" + quadrado.getX() +  
                            "," + quadrado.getY() +  
                            ") - Lado: " + quadrado.getLado() );  
    }  
}
```

## Exemplo — Formas Geométricas

- As classes **Circulo** e **Quadrado** têm em comum alguns dos atributos e métodos (código duplicado):
  - 2 atributos (x e y)
  - 5 *getters* & *setters*
  
- A solução é utilizar a herança criando uma superclasse **FormaGeometrica** e definindo **Circulo** e **Quadrado** como Subclasses.

# Exemplo — Formas Geométricas



# Exemplo — Formas Geométricas

```
public class FormaGeometrica {  
    private int x, y;  
  
    public FormaGeometrica () {  
        x = 0;  
        y = 0;  
    }  
  
    public FormaGeometrica (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
  
    public void deslocar( int dx, int dy ) {  
        x += dx; y += dy;  
    }  
}
```

# Exemplo — Formas Geométricas

```
public class Circulo extends FormaGeometrica {  
    private int raio;  
  
    public Circulo() {  
        super(0, 0);  
        this.raio = 1;  
    }  
  
    public Circulo(int x, int y, int raio) {  
        super(x, y);  
        this.raio = raio;  
    }  
  
    public int getRaio() {  
        return raio;  
    }  
  
    public void setRaio(int raio) {  
        this.raio = raio;  
    }  
}
```

Acrescenta apenas o atributo **raio** e os métodos seletores e modificadores associados

# Exemplo — Formas Geométricas

```
public class Quadrado extends FormaGeometrica {  
    private int lado;  
  
    public Quadrado() {  
        super(0, 0);  
        this.lado = 1;  
    }  
  
    public Quadrado(int x, int y, int lado) {  
        super(x, y);  
        this.lado = lado;  
    }  
  
    public int getLado() {  
        return lado;  
    }  
  
    public void setLado(int lado) {  
        this.lado = lado;  
    }  
}
```

Acrescenta apenas o atributo **lado** e os métodos seletores e modificadores associados

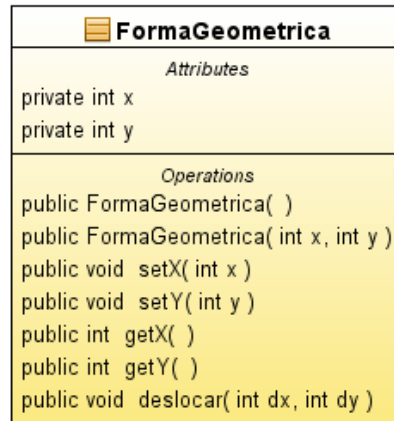
# Exemplo — Formas Geométricas

O método main do programa não sofre qualquer alteração

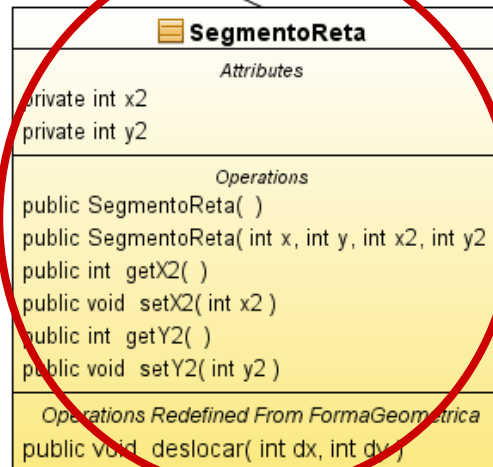
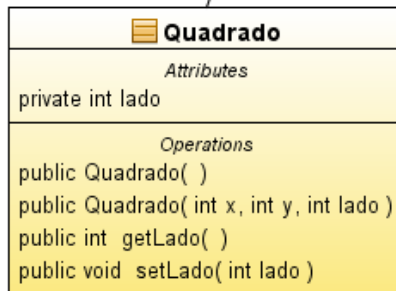
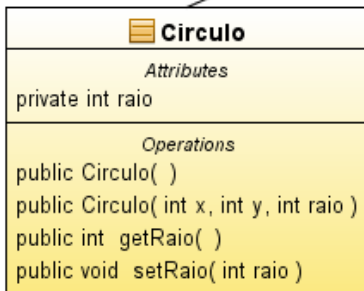
```
public class Programa {  
    public static void main(String[] args) {  
        Circulo circulo = new Circulo(1, 1, 23);  
        Quadrado quadrado = new Quadrado(0, 0, 4);  
        System.out.println("Circulo: - Posição (" + circulo.getX() +  
            "," + circulo.getY() +  
            ") - Raio: " + circulo.getRaio() );  
        System.out.println("Quadrado: - Posição (" + quadrado.getX() +  
            "," + quadrado.getY() +  
            ") - Lado: " + quadrado.getLado() );  
        quadrado.deslocar( 2, 2);  
        System.out.println("Quadrado: - Posição (" + quadrado.getX() +  
            "," + quadrado.getY() +  
            ") - Lado: " + quadrado.getLado() );  
    }  
}
```

**E se quisermos criar uma  
nova classe para representar  
um segmento de reta?**

# Exemplo — Formas Geométricas



Nova classe definida como  
**subclasse de**  
**FormaGeometrica**





# Exemplo — Formas Geométricas

```
public class SegmentoReta extends FormaGeometrica {  
    private int x2, y2;  
    public SegmentoReta() {  
        super(0, 0);  
        this.x2 = 1;  
        this.y2 = 1;  
    }  
    public SegmentoReta(int x, int y, int x2, int y2) {  
        super(x, y);  
        this.x2 = x2;  
        this.y2 = y2;  
    }  
    public int getX2() {  
        return x2;  
    }  
    public void setX2(int x2) {  
        this.x2 = x2;  
    }  
    public int getY2() {  
        return y2;  
    }  
    public void setY2(int y2) {  
        this.y2 = y2;  
    }  
}
```

Acrescenta os atributos **x2** e **y2** para a representação do segundo ponto do segmento de reta

**Existe um método que é herdado mas não é adequado nesta classe. Qual?**

# Herança — Redefinição de Métodos

## □ O método:

```
public void deslocar( int dx, int dy ) {  
    x += dx; y += dy;  
}
```

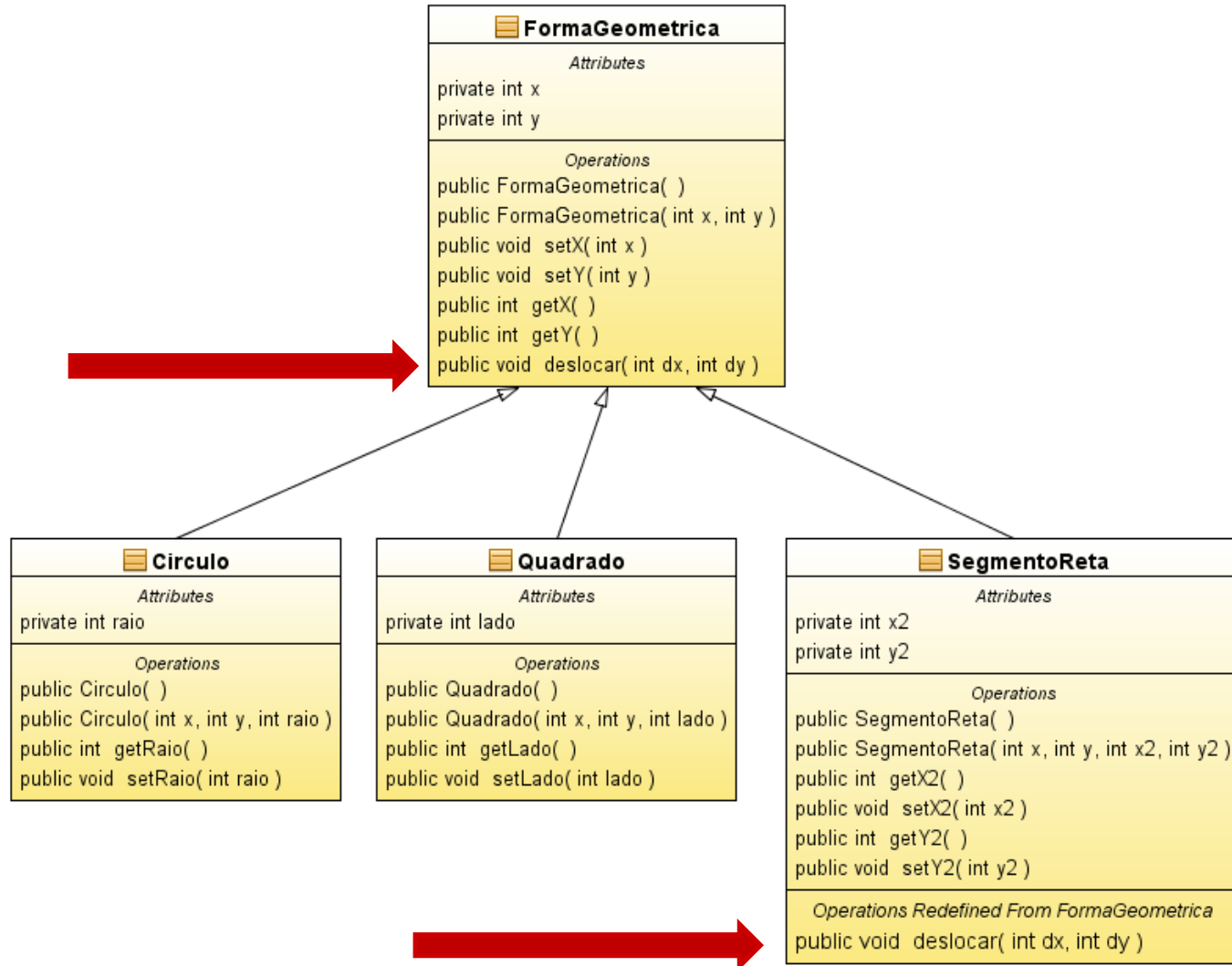
- Este método não funciona corretamente para objetos da classe **SegmentoReta**.
- A solução é redefinir este método nesta classe.

■ Para redefinir um método **basta defini-lo novamente** no corpo da subclasse:



```
public class SegmentoRecta extends FormaGeometrica {  
    // código omitido  
  
    @Override  
    public void deslocar( int dx, int dy ) {  
        super.deslocar(dx, dy);  
        x2 += dx; y2 += dy;  
    }  
}
```

# Exemplo — Formas Geométricas



# Herança (is-a) — Generalização vs Especialização

- Uma **superclasse** de outras classes representa a **generalização** dessas classes
  - A superclasse é mais genérica que as subclasses
    - Por exemplo um **Veiculo** é uma generalização de **Mota** e **Automovel**.

Versus ...

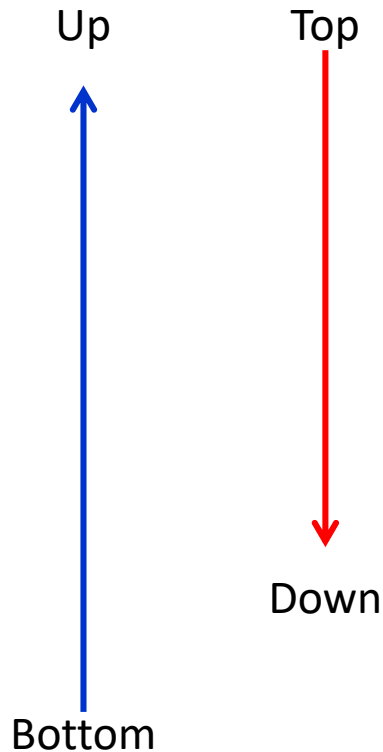
- Uma **subclasse** de uma dada classe é uma **especialização** dessa classe
  - As subclasses especializam a sua superclasse
    - A classe **Cao** é uma especialização da classe **Animal**.

# Herança (is-a) — Generalização vs Especialização

- Na prática a herança utiliza-se por **generalização** e por **especialização** de classes

- **Generalização**

- Quando, numa aplicação, se verifica que várias classes partilham um conjunto de atributos e/ou métodos e que a relação de herança pode ser aplicada com a criação duma classe mais genérica que faça sentido dizemos que estamos a usar herança por generalização

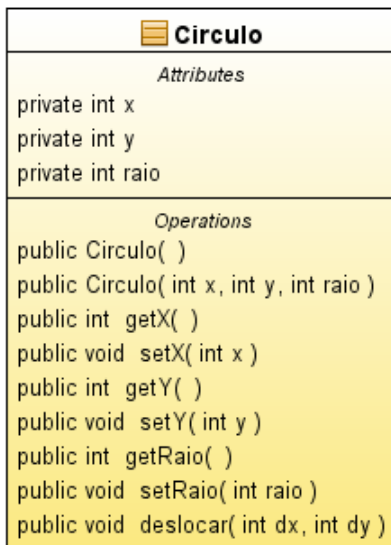
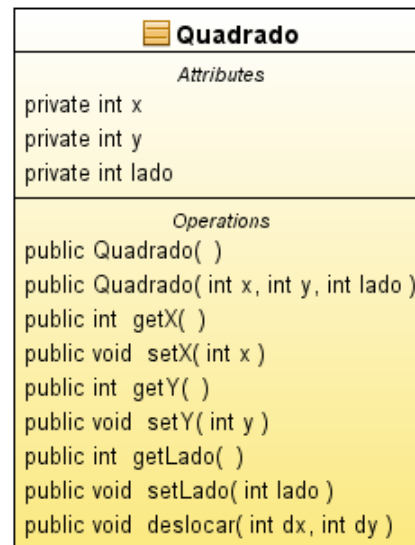
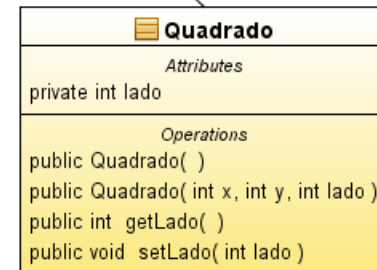
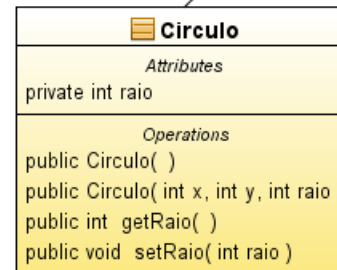
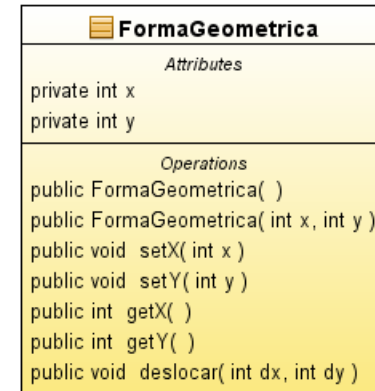


- **Especialização**

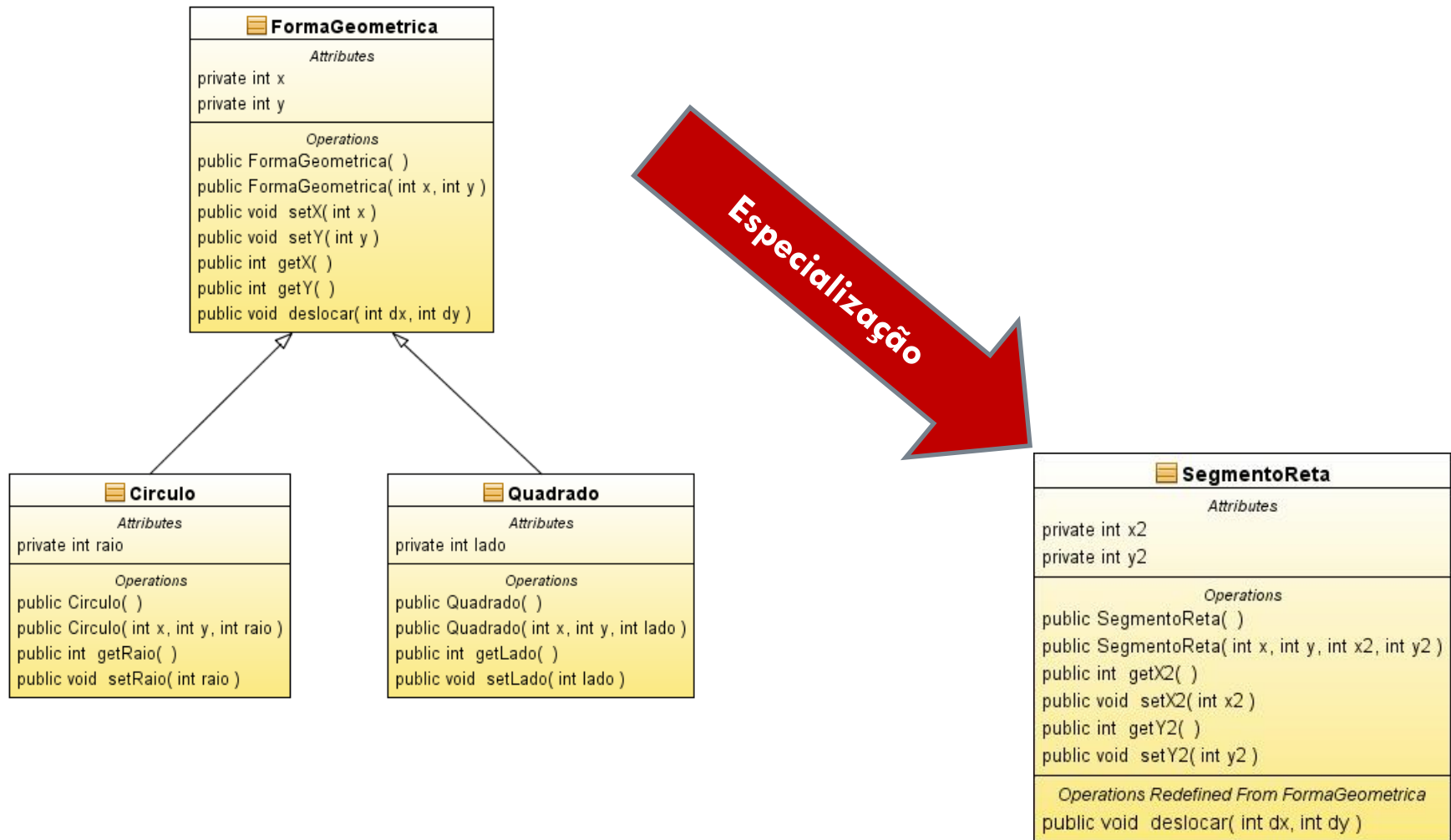
- Quando já existe classe genérica e se verifica que a classe que se pretende criar pode ser derivada por herança dessa classe e onde essa herança faz sentido, dizemos que estamos a usar a herança por especialização

# Herança (is-a) — Generalização vs Especialização

Generalização




# Herança (is-a) — Generalização vs Especialização



# Classe Object

- A classe **Object** é uma classe do Java, que está no topo da hierarquia e da qual todas as outras são subclasses diretas ou indiretas.
  - Todas as classes são uma especialização de **Object**, são todas um tipo de *objeto*.
  - Quando uma classe não deriva de outra o compilador de Java coloca-a a derivar de **Object** (é acrescentado *extends Object*).
  - A classe **Object** define um conjunto de métodos que são herdados por todas as classes, entre eles estão os métodos: **toString**, **equals** e **hashCode** utilizados antes.
    - Por isso quando se coloca um destes métodos numa classe, na prática está-se a redefinir o método herdado, sendo então necessário colocar **@Override** antes da redefinição

|  <b>Object</b>  |
|--|
| <i>Attributes</i>  |
| <i>Operations</i><br>public Object( )<br>public Class getClass( )<br>public int hashCode( )<br>public boolean equals( Object o )<br>public String toString( )<br>public void notify( )<br>public void notifyAll( )<br>public void wait( long l )<br>public void wait( long l, int i )<br>public void wait( ) |



# Bibliografia

- Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - Capítulo 10

