Aula 21 – Métodos Abstratos e Interfaces

Norton Trevisan Roman

13 de junho de 2013

Métodos e Classes Abstratos

- Vamos rever *Casa*

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

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?

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

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?
  - Para que pudessemos trabalhar com objetos de subclasses em código que exige a superclasse

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

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?
  - Para que pudessemos trabalhar com objetos de subclasses em código que exige a superclasse
  - Como em *Residencia*

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

class Residencia {
    Casa casa;
    AreaPiscina piscina;
    public double area() {
        double resp = 0;
        if (this.casa != null)
            resp += this.casa.area();
        if (this.piscina != null)
            resp += this.piscina.area();
        return(resp);
    }
}
```

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?
  - Para que pudessemos trabalhar com objetos de subclasses em código que exige a superclasse
  - Como em *Residencia*
- Bem coxambrado

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

class Residencia {
    Casa casa;
    AreaPiscina piscina;
    public double area() {
        double resp = 0;
        if (this.casa != null)
            resp += this.casa.area();
        if (this.piscina != null)
            resp += this.piscina.area();
        return(resp);
    }
}
```

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?
  - Para que pudessemos trabalhar com objetos de subclasses em código que exige a superclasse
  - Como em *Residencia*
- Bem coxambrado
- Como fazer então?

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

class Residencia {
    Casa casa;
    AreaPiscina piscina;
    public double area() {
        double resp = 0;
        if (this.casa != null)
            resp += this.casa.area();
        if (this.piscina != null)
            resp += this.piscina.area();
        return(resp);
    }
}
```

Métodos e Classes Abstratos

- Vamos rever *Casa*
- Por que fizemos isso?
  - Para que pudessemos trabalhar com objetos de subclasses em código que exige a superclasse
  - Como em *Residencia*
- Bem coxambrado
- Como fazer então?
  - Tornar o método abstrato

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

class Residencia {
    Casa casa;
    AreaPiscina piscina;
    public double area() {
        double resp = 0;
        if (this.casa != null)
            resp += this.casa.area();
        if (this.piscina != null)
            resp += this.piscina.area();
        return(resp);
    }
}
```

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?
  - As subclasses

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?
  - As subclasses
  - Subclasses são obrigadas a implementar métodos abstratos da superclasse

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?
  - As subclasses
  - Subclasses são obrigadas a implementar métodos abstratos da superclasse
- A existência de métodos abstratos torna a classe abstrata

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

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?
  - As subclasses
  - Subclasses são obrigadas a implementar métodos abstratos da superclasse

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

- A existência de métodos abstratos torna a classe abstrata
- Quando aplicado a classes, *abstract* faz com que não possam ser instanciadas

Métodos e Classes Abstratos

- São métodos sem uma implementação definida na classe
  - Possuem apenas o necessário para compilar: sua assinatura
- Quem os implementa então?
  - As subclasses
  - Subclasses são obrigadas a implementar métodos abstratos da superclasse

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

- A existência de métodos abstratos torna a classe abstrata
- Quando aplicado a classes, *abstract* faz com que não possam ser instanciadas
  - Não podemos fazer *new Classe()*

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract

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

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract

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

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

- Nesse caso, *Casa* foi escolhida como abstrata pelo fato de, neste sistema, existirem apenas casas quadradas e retangulares.

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

- Nesse caso, *Casa* foi escolhida como abstrata pelo fato de, neste sistema, existirem apenas casas quadradas e retangulares.
  - Não deveria ser possível criar uma casa genérica

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

- Nesse caso, *Casa* foi escolhida como abstrata pelo fato de, neste sistema, existirem apenas casas quadradas e retangulares.
  - Não deveria ser possível criar uma casa genérica
  - Por isso usamos abstract

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

- Nesse caso, *Casa* foi escolhida como abstrata pelo fato de, neste sistema, existirem apenas casas quadradas e retangulares.
  - Não deveria ser possível criar uma casa genérica
  - Por isso usamos abstract
- E podemos ter parâmetros em métodos abstratos?

Métodos e Classes Abstratos

- Todo método abstract torna a classe abstract
- Porém nem toda classe abstract possui métodos abstract
  - Basta que não desejemos que seja instanciada

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

- Nesse caso, *Casa* foi escolhida como abstrata pelo fato de, neste sistema, existirem apenas casas quadradas e retangulares.
  - Não deveria ser possível criar uma casa genérica
  - Por isso usamos abstract
- E podemos ter parâmetros em métodos abstratos?
  - Sim. Nesse exemplo não foi preciso, mas o método abstrato é idêntico à assinatura de sua versão normal, exceto pelo modificador *abstract*

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:
  - Quiser impedir a existência de objetos dela

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:
  - Quiser impedir a existência de objetos dela
    - ★ Somente as subclasses poderão ter

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:
  - Quiser impedir a existência de objetos dela
    - ★ Somente as subclasses poderão ter
  - Necessitar que algum método seja conhecido na superclasse

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:
  - Quiser impedir a existência de objetos dela
    - ★ Somente as subclasses poderão ter
  - Necessitar que algum método seja conhecido na superclasse
    - ★ Mas que não possa/precise ter código nela

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Métodos e Classes Abstratos

- E como fica o código das subclasses de *Casa*?
  - Idênticos. Nada muda. Apenas consertamos um coxambre.
- Em suma, use classes abstratas quando:
  - Quiser impedir a existência de objetos dela
    - ★ Somente as subclasses poderão ter
  - Necessitar que algum método seja conhecido na superclasse
    - ★ Mas que não possa/precise ter código nela
    - ★ Apenas faz sentido o código nas subclasses

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

    CasaQuad() {}

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

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

    public double area() {
        double areat=-1; // área total

        if (this.lateral>=0) {
            areat = this.lateral*this.lateral;
        }
        return(areat);
    }
}
```

Interfaces

- Voltemos agora ao método da bolha (ordenação), usado em *Projeto*

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos agora ao método da bolha (ordenação), usado em *Projeto*
- Depende do método *comparaRes* de *Residencia*

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos agora ao método da bolha (ordenação), usado em *Projeto*
- Depende do método *comparaRes* de *Residencia*
  - Compara as residências pela área

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }
}
```

Interfaces

- E se quisermos comparar por valor?

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }




}
```

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores
  - Além de mudar o método de ordenação (e alguns especificadores em outras classes)

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores
  - Além de mudar o método de ordenação (e alguns especificadores em outras classes)

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- E se quisermos comparar por valor?
  - Teremos que implementar cada um dos comparadores
  - Além de mudar o método de ordenação (e alguns especificadores em outras classes)

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}

class Projeto {
    ...
    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- O que há de errado com o código?

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

```
class Projeto {
    ...
    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- O que há de errado com o código?
  - Muita duplicidade

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

```
class Projeto {
    ...
    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- O que há de errado com o código?
  - Muita duplicidade
  - Trabalho duplicado

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}

class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}

class Projeto {
    ...
    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- O que há de errado com o código?
  - Muita duplicidade
  - Trabalho duplicado
  - Problemas na hora de dar manutenção

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

```
class Residencia {
    ...
    public int comparaRes(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.area() - outra.area()));
    }

    public int comparaResP(Residencia outra) {
        if (outra == null) return(1);
        return((int)(this.casa.valor(this.casa.area())
              - outra.casa.valor(outra.casa.area())));
    }
}
```

```
class Projeto {
    ...
    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Como resolver?

Interfaces

- Como resolver?
  - Interfaces

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*
  - Não podemos criar objetos das interfaces

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*
  - Não podemos criar objetos das interfaces

```
interface Teste {
    int x(int y);
    void y();
}
```

```
class Testando {
    public static void main(
                    String[] args) {
        Teste t = new Teste();
    }
}
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*
  - Não podemos criar objetos das interfaces

```
interface Teste {
    int x(int y);
    void y();
}
```

```
class Testando {
    public static void main(
                    String[] args) {
        Teste t = new Teste();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:4: Teste is abstract; cannot
be instantiated
Teste t = new Teste();
          ^
1 error
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*
  - Não podemos criar objetos das interfaces
- São como classes abstratas em que todos os métodos são abstratos

```
interface Teste {
    int x(int y);
    void y();
}
```

```
class Testando {
    public static void main(
                    String[] args) {
        Teste t = new Teste();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:4: Teste is abstract; cannot
be instantiated
Teste t = new Teste();
          ^
1 error
```

Interfaces

- Como resolver?
  - Interfaces
- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - ★ Não precisamos usar *public*
  - Não podemos criar objetos das interfaces
- São como classes abstratas em que todos os métodos são abstratos
  - Por isso, não há *static* em métodos de interfaces

```
interface Teste {
    int x(int y);
    void y();
}
```

```
class Testando {
    public static void main(
                    String[] args) {
        Teste t = new Teste();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:4: Teste is abstract; cannot
be instantiated
Teste t = new Teste();
          ^
1 error
```

Interfaces

- Se nao podemos criar objetos, como usá-las?

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {



}
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {



}
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses
- Vai compilar?

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {



}
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses
- Vai compilar?

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {



}
```

Saída

```
$ javac Testando.java
Testando.java:1: Testando is not abstract
and does not override abstract method y()
in Teste
class Testando implements Teste {
^
1 error
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses
- Vai compilar?
  - Assim como classes abstratas exigem que suas subclasses implementem seus métodos abstratos, também interfaces exigem que as classes que as implementam implementem os métodos

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {



}
```

Saída

```
$ javac Testando.java
Testando.java:1: Testando is not abstract
and does not override abstract method y()
in Teste
class Testando implements Teste {
^
1 error
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses
- Vai compilar?
  - Assim como classes abstratas exigem que suas subclasses implementem seus métodos abstratos, também interfaces exigem que as classes que as implementam implementem os métodos

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    int x(int y) {
        return(y);
    }
    void y() {
        System.out.println();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:1: Testando is not abstract
and does not override abstract method y()
in Teste
class Testando implements Teste {
^
1 error
```

Interfaces

- Se nao podemos criar objetos, como usá-las?
  - Necessitam de classes que as implementem
  - Como o *extends* de subclasses
- Vai compilar?
  - Assim como classes abstratas exigem que suas subclasses implementem seus métodos abstratos, também interfaces exigem que as classes que as implementam implementem os métodos
  - São forçadas a prover código para os métodos das interfaces.

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    int x(int y) {
        return(y);
    }
    void y() {
        System.out.println();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:1: Testando is not abstract
and does not override abstract method y()
in Teste
class Testando implements Teste {
^
1 error
```

Interfaces

- E agora? Vai compilar?

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    int x(int y) {
        return(y);
    }
    void y() {
        System.out.println();
    }
}
```

Interfaces

- E agora? Vai compilar?

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    int x(int y) {
        return(y);
    }
    void y() {
        System.out.println();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:6: y() in Testando cannot implement y() in Teste; attempting to assign
weaker access privileges; was public
    void y() {
         ^
Testando.java:2: x(int) in Testando cannot implement x(int) in Teste; attempting to
assign weaker access privileges; was public
    int x(int y) {
        ^
2 errors
```

Interfaces

- E agora? Vai compilar?
- Lembre-se que métodos em uma interface são implicitamente públicos

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    int x(int y) {
        return(y);
    }
    void y() {
        System.out.println();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:6: y() in Testando cannot implement y() in Teste; attempting to assign
weaker access privileges; was public
    void y() {
         ^
Testando.java:2: x(int) in Testando cannot implement x(int) in Teste; attempting to
assign weaker access privileges; was public
    int x(int y) {
        ^
2 errors
```

Interfaces

- E agora? Vai compilar?
- Lembre-se que métodos em uma interface são implicitamente públicos

```
interface Teste {
    int x(int y);
    void y();
}

class Testando implements Teste {
    public int x(int y) {
        return(y);
    }
    public void y() {
        System.out.println();
    }
}
```

Saída

```
$ javac Testando.java
Testando.java:6: y() in Testando cannot implement y() in Teste; attempting to assign
weaker access privileges; was public
    void y() {
         ^
Testando.java:2: x(int) in Testando cannot implement x(int) in Teste; attempting to
assign weaker access privileges; was public
    int x(int y) {
        ^
2 errors
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe

```
interface Teste {
    int x(int y);
    void y();
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe

```
interface Teste {
    int x(int y);
    void y();
}

public class Minha {
    public int calculo(Teste t, int y) {
        return(2*t.x(y));
    }
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe
  - Como quando usávamos a superclasse em lugar das subclasses

```
interface Teste {
    int x(int y);
    void y();
}

public class Minha {
    public int calculo(Teste t, int y) {
        return(2*t.x(y));
    }
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe
  - Como quando usávamos a superclasse em lugar das subclasses
  - Como não há como criar um objeto *Teste*, quem quiser usar a classe *Minha* terá que criar uma classe que implemente *Teste* e usá-la no lugar

```
interface Teste {
    int x(int y);
    void y();
}

public class Minha {
    public int calculo(Teste t, int y) {
        return(2*t.x(y));
    }
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe
  - Como quando usávamos a superclasse em lugar das subclasses
  - Como não há como criar um objeto *Teste*, quem quiser usar a classe *Minha* terá que criar uma classe que implemente *Teste* e usá-la no lugar

```
interface Teste {
    int x(int y);
    void y();
}

public class Minha {
    public int calculo(Teste t, int y) {
        return(2*t.x(y));
    }
}

class Testando implements Teste {
    public int x(int y) {
        return(y);
    }

    public void y() {
        System.out.println();
    }

    public static void main(String[] args) {
        Minha m = new Minha();

        System.out.println(m.calculo(
                            new Testando(),2));
    }
}
```

Interfaces

- Particularmente úteis quando queremos forçar o programador a manter a assinatura de métodos, para usar classes por nós desenvolvidas
  - Para isso, basta usar a interface como se fosse uma classe
  - Como quando usávamos a superclasse em lugar das subclasses
  - Como não há como criar um objeto *Teste*, quem quiser usar a classe *Minha* terá que criar uma classe que implemente *Teste* e usá-la no lugar
- Ideal para trabalhos em equipe

```
interface Teste {
    int x(int y);
    void y();
}

public class Minha {
    public int calculo(Teste t, int y) {
        return(2*t.x(y));
    }
}

class Testando implements Teste {
    public int x(int y) {
        return(y);
    }

    public void y() {
        System.out.println();
    }

    public static void main(String[] args) {
        Minha m = new Minha();

        System.out.println(m.calculo(
                            new Testando(),2));
    }
}
```

Interfaces

- Voltemos ao nosso código

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos ao nosso código
- A única coisa que muda é o modo de compararmos os objetos *Residencia*

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos ao nosso código
- A única coisa que muda é o modo de compararmos os objetos *Residencia*
- Seria interessante termos uma espécie de comparador universal, que impedisse essa duplicidade de código

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos ao nosso código
- A única coisa que muda é o modo de compararmos os objetos *Residencia*
- Seria interessante termos uma espécie de comparador universal, que impedisse essa duplicidade de código
  - Assim, se quiséssemos comparar por área, usaríamos sua versão área

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos ao nosso código
- A única coisa que muda é o modo de compararmos os objetos *Residencia*
- Seria interessante termos uma espécie de comparador universal, que impedisse essa duplicidade de código
  - Assim, se quiséssemos comparar por área, usaríamos sua versão área
  - Se quiséssemos comparar valor, usaríamos sua versão valor

```
class Projeto {
    ...
    static void bolha(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaRes(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

    static void bolhaP(Residencia[] v) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (v[i].comparaResP(v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
```

Interfaces

- Voltemos ao nosso código
- A única coisa que muda é o modo de compararmos os objetos *Residencia*
- Seria interessante termos uma espécie de comparador universal, que impedisse essa duplicidade de código
  - Assim, se quiséssemos comparar por área, usaríamos sua versão área
  - Se quiséssemos comparar valor, usaríamos sua versão valor

```
class Projeto {
    ...
    static void bolha(Residencia[] v, Comparador c) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (c.compara(v[i],v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }

}
```

Interfaces

- Então...

Interfaces

- Então...

```
interface Comparador {
    int compara(Residencia a, Residencia b);
}
```

Interfaces

- Então...
- Também temos que implementar os comparadores

```
interface Comparador {
    int compara(Residencia a, Residencia b);
}
```

Interfaces

- Então...
- Também temos que implementar os comparadores

```
interface Comparador {
    int compara(Residencia a, Residencia b);
}

class ComparaArea implements Comparador {
    public int compara(Residencia a, Residencia b)
    {
        if (a == null) return(-1);
        if (b == null) return(1);
        return((int)(a.area() - b.area()));
    }
}

class ComparaValor implements Comparador {
    public int compara(Residencia a, Residencia b)
    {
        if (a == null) return(-1);
        if (b == null) return(1);
        return((int)(a.casa.valor(a.casa.area()) -
                     b.casa.valor(b.casa.area())));
    }
}
```

Interfaces

- Então...
- Também temos que implementar os comparadores
- Deverão substituir os métodos *comparaRes* e *comparaRes* de *Residencia*

```
interface Comparador {
    int compara(Residencia a, Residencia b);
}

class ComparaArea implements Comparador {
    public int compara(Residencia a, Residencia b)
    {
        if (a == null) return(-1);
        if (b == null) return(1);
        return((int)(a.area() - b.area()));
    }
}

class ComparaValor implements Comparador {
    public int compara(Residencia a, Residencia b)
    {
        if (a == null) return(-1);
        if (b == null) return(1);
        return((int)(a.casa.valor(a.casa.area()) -
                     b.casa.valor(b.casa.area())));
    }
}
```

Interfaces

- E como usamos isso?

Interfaces

- E como usamos 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("Área r1: "+r1.area());
    System.out.println("Área r2: "+r2.area());
    Comparador c = new ComparaArea();
    System.out.println("Comparação: "+
                                    c.compara(r1,r2));

    System.out.println();

    System.out.println("Valor casa r1: "+
                    r1.casa.valor(r1.casa.area()));
    System.out.println("Valor casa r2: "+
                    r2.casa.valor(r2.casa.area()));
    c = new ComparaValor();
    System.out.println("Comparação: "+
                                     c.compara(r1,r2));
}
```

Interfaces

- E como usamos isso?

Saída

```
$ java Projeto
Área r1: 150.0
Área r2: 100.0
Comparação: 50

Valor casa r1: 198000.0
Valor casa r2: 152300.0
Comparação: 45700
```

```
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("Área r1: "+r1.area());
    System.out.println("Área r2: "+r2.area());
    Comparador c = new ComparaArea();
    System.out.println("Comparação: "+
                                    c.compara(r1,r2));

    System.out.println();

    System.out.println("Valor casa r1: "+
                    r1.casa.valor(r1.casa.area()));
    System.out.println("Valor casa r2: "+
                    r2.casa.valor(r2.casa.area()));
    c = new ComparaValor();
    System.out.println("Comparação: "+
                                    c.compara(r1,r2));
}
```

Interfaces

- Ex: sabemos 3 métodos de ordenação

Interfaces

- Ex: sabemos 3 métodos de ordenação
  - Bolha,

```
static void bolha(Residencia[] v, Comparador c) {
    for (int ult = v.length-1; ult>0; ult--) {
        for (int i=0; i<ult; i++) {
            if (c.compara(v[i],v[i+1]) > 0) {
                Residencia aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
            }
        }
    }
}
```

Interfaces

- Ex: sabemos 3 métodos de ordenação
  - Bolha, seleção

```
static int posMenorEl(Residencia[] v, int inicio,
                      int fim, Comparador c) {
    int posMenor = -1;
    if ((v!=null) && (inicio>=0) &&
          (fim <= v.length) && (inicio < fim)) {
        posMenor = inicio;
        for (int i=inicio+1; i<fim; i++) {
            if (c.compara(v[i],v[posMenor]) < 0)
                                posMenor = i;
        }
    }
    return(posMenor);
}
static void selecao(Residencia[] v, Comparador c) {
    for (int i=0; i<v.length-1; i++) {
        int posMenor = posMenorEl(v,i,v.length,c);
        if (c.compara(v[posMenor],v[i]) < 0) {
            Residencia aux = v[i];
            v[i] = v[posMenor];
            v[posMenor] = aux;
        }
    }
}
```

```
static void bolha(Residencia[] v, Comparador c) {
    for (int ult = v.length-1; ult>0; ult--) {
        for (int i=0; i<ult; i++) {
            if (c.compara(v[i],v[i+1]) > 0) {
                Residencia aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
            }
        }
    }
}
```

Interfaces

- Ex: sabemos 3 métodos de ordenação
  - Bolha, seleção e inserção

```
static int posMenorEl(Residencia[] v, int inicio,
                      int fim, Comparador c) {
    int posMenor = -1;
    if ((v!=null) && (inicio>=0) &&
          (fim <= v.length) && (inicio < fim)) {
        posMenor = inicio;
        for (int i=inicio+1; i<fim; i++) {
            if (c.compara(v[i],v[posMenor]) < 0)
                                posMenor = i;
        }
    }
    return(posMenor);
}
static void selecao(Residencia[] v, Comparador c) {
    for (int i=0; i<v.length-1; i++) {
        int posMenor = posMenorEl(v,i,v.length,c);
        if (c.compara(v[posMenor],v[i]) < 0) {
            Residencia aux = v[i];
            v[i] = v[posMenor];
            v[posMenor] = aux;
        }
    }
}
```

```
static void bolha(Residencia[] v, Comparador c) {
    for (int ult = v.length-1; ult>0; ult--) {
        for (int i=0; i<ult; i++) {
            if (c.compara(v[i],v[i+1]) > 0) {
                Residencia aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
            }
        }
    }
}

static void insercao(Residencia[] v,Comparador c){
    for (int i=1; i<v.length; i++) {
        Residencia aux = v[i];
        int j = i;
        while ((j > 0) &&
                    (c.compara(aux,v[j-1]) < 0)) {
            v[j] = v[j-1];
            j--;
        }
        v[j] = aux;
    }
}
```

Interfaces

- Ex: sabemos 3 métodos de ordenação
  - Bolha, seleção e inserção

```
static int posMenorEl(Residencia[] v, int inicio,
                      int fim, Comparador c) {
    int posMenor = -1;
    if ((v!=null) && (inicio>=0) &&
          (fim <= v.length) && (inicio < fim)) {
        posMenor = inicio;
        for (int i=inicio+1; i<fim; i++) {
            if (c.compara(v[i],v[posMenor]) < 0)
                                    posMenor = i;
        }
    }
    return(posMenor);
}
static void selecao(Residencia[] v, Comparador c) {
    for (int i=0; i<v.length-1; i++) {
        int posMenor = posMenorEl(v,i,v.length,c);
        if (c.compara(v[posMenor],v[i]) < 0) {
            Residencia aux = v[i];
            v[i] = v[posMenor];
            v[posMenor] = aux;
        }
    }
}
```

```
static void bolha(Residencia[] v, Comparador c) {
    for (int ult = v.length-1; ult>0; ult--) {
        for (int i=0; i<ult; i++) {
            if (c.compara(v[i],v[i+1]) > 0) {
                Residencia aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
            }
        }
    }
}

static void insercao(Residencia[] v,Comparador c){
    for (int i=1; i<v.length; i++) {
        Residencia aux = v[i];
        int j = i;
        while ((j > 0) &&
                    (c.compara(aux,v[j-1]) < 0)) {
            v[j] = v[j-1];
            j--;
        }
        v[j] = aux;
    }
}
```

Note que usamos *Comparador*

Interfaces

- Qual o problema?

Interfaces

- Qual o problema?
  - O usuário (programador), no código dele, terá que explicitamente optar por um dos ordenadores

```
public static void main(String[] args) {
    Residencia[] cond = new Residencia[5];

    for (int i=0; i<5; i++) {
        CasaQuad c = new CasaQuad(Math.random()
                                              *100);
        Residencia r = new Residencia(c,null);
        cond[i] = r;
    }

    for (Residencia r : cond)
        System.out.println(r.area());
    System.out.println();

    Comparador c = new ComparaArea();
    bolha(cond,c);

    for (Residencia r : cond)
        System.out.println(r.area());
}
```

Interfaces

- Qual o problema?
  - O usuário (programador), no código dele, terá que explicitamente optar por um dos ordenadores
- Que fazer?

```
public static void main(String[] args) {
    Residencia[] cond = new Residencia[5];

    for (int i=0; i<5; i++) {
        CasaQuad c = new CasaQuad(Math.random()
                                              *100);
        Residencia r = new Residencia(c,null);
        cond[i] = r;
    }

    for (Residencia r : cond)
        System.out.println(r.area());
    System.out.println();

    Comparador c = new ComparaArea();
    bolha(cond,c);

    for (Residencia r : cond)
        System.out.println(r.area());
}
```

Interfaces

- Qual o problema?
  - O usuário (programador), no código dele, terá que explicitamente optar por um dos ordenadores
- Que fazer?
  - Antes de mais nada, transformamos cada método em uma classe diferente

```
public static void main(String[] args) {
    Residencia[] cond = new Residencia[5];

    for (int i=0; i<5; i++) {
        CasaQuad c = new CasaQuad(Math.random()
                                              *100);
        Residencia r = new Residencia(c,null);
        cond[i] = r;
    }

    for (Residencia r : cond)
        System.out.println(r.area());
    System.out.println();

    Comparador c = new ComparaArea();
    bolha(cond,c);

    for (Residencia r : cond)
        System.out.println(r.area());
}
```

Interfaces

- Qual o problema?
  - O usuário (programador), no código dele, terá que explicitamente optar por um dos ordenadores
- Que fazer?
  - Antes de mais nada, transformamos cada método em uma classe diferente
  - Implementando uma interface comum a todos

```
public static void main(String[] args) {
    Residencia[] cond = new Residencia[5];

    for (int i=0; i<5; i++) {
        CasaQuad c = new CasaQuad(Math.random()
                                              *100);
        Residencia r = new Residencia(c,null);
        cond[i] = r;
    }

    for (Residencia r : cond)
        System.out.println(r.area());
    System.out.println();

    Comparador c = new ComparaArea();
    bolha(cond,c);

    for (Residencia r : cond)
        System.out.println(r.area());
}
```

Interfaces

```
public class Selecao implements Ordenador {
    public int posMenorEl(Residencia[] v,
           int inicio, int fim, Comparador c) {
        int posMenor = -1;
        if ((v!=null) && (inicio>=0) &&
         (fim <= v.length) && (inicio < fim)) {
            posMenor = inicio;
            for (int i=inicio+1; i<fim; i++) {
                if (c.compara(v[i],v[posMenor])
                                < 0) posMenor = i;
            }
        }
        return(posMenor);
    }
    public void ordena(Residencia[] v,
                                Comparador c) {
        for (int i=0; i<v.length-1; i++) {
            int posMenor =
                      posMenorEl(v,i,v.length,c);
            if (c.compara(v[posMenor],v[i])
                                            < 0) {
                Residencia aux = v[i];
                v[i] = v[posMenor];
                v[posMenor] = aux;
            }
        }
    }
}
```

```
public class Bolha implements Ordenador {
    public void ordena(Residencia[] v,Comparador c) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (c.compara(v[i],v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
public class Insercao implements Ordenador {
    public void ordena(Residencia[] v,Comparador c) {
        for (int i=1; i<v.length; i++) {
            Residencia aux = v[i];
            int j = i;
            while ((j > 0) &&
                        (c.compara(aux,v[j-1]) < 0)) {
                v[j] = v[j-1];
                j--;
            }
            v[j] = aux;
        }
    }
}
```

Interfaces

```
public class Selecao implements Ordenador {
    public int posMenorEl(Residencia[] v,
           int inicio, int fim, Comparador c) {
        int posMenor = -1;
        if ((v!=null) && (inicio>=0) &&
         (fim <= v.length) && (inicio < fim)) {
            posMenor = inicio;
            for (int i=inicio+1; i<fim; i++) {
                if (c.compara(v[i],v[posMenor])
                                < 0) posMenor = i;
            }
        }
        return(posMenor);
    }
    public void ordena(Residencia[] v,
                                Comparador c) {
        for (int i=0; i<v.length-1; i++) {
            int posMenor =
                      posMenorEl(v,i,v.length,c);
            if (c.compara(v[posMenor],v[i])
                                            < 0) {
                Residencia aux = v[i];
                v[i] = v[posMenor];
                v[posMenor] = aux;
            }
        }
    }
}
```

```
public class Bolha implements Ordenador {
    public void ordena(Residencia[] v,Comparador c) {
        for (int ult = v.length-1; ult>0; ult--) {
            for (int i=0; i<ult; i++) {
                if (c.compara(v[i],v[i+1]) > 0) {
                    Residencia aux = v[i];
                    v[i] = v[i+1];
                    v[i+1] = aux;
                }
            }
        }
    }
}
public class Insercao implements Ordenador {
    public void ordena(Residencia[] v,Comparador c) {
        for (int i=1; i<v.length; i++) {
            Residencia aux = v[i];
            int j = i;
            while ((j > 0) &&
                        (c.compara(aux,v[j-1]) < 0)) {
                v[j] = v[j-1];
                j--;
            }
            v[j] = aux;
        }
    }
}
```

- Note que os métodos não são mais *static* (por conta da interface)

Interfaces

- E como fica a interface propriamente dita?

Interfaces

- E como fica a interface propriamente dita?

```
public interface Ordenador {
    void ordena(Residencia[] v,
                        Comparador c);
}
```

Interfaces

- E como fica a interface propriamente dita?

```
public interface Ordenador {
    void ordena(Residencia[] v,
                        Comparador c);
}
```

- E o código que faz a chamada?

Interfaces

- E como fica a interface propriamente dita?

```
public interface Ordenador {
    void ordena(Residencia[] v,
                        Comparador c);
}
```

- E o código que faz a chamada?

```
class X {
    public static void main(String[] args) {
        Residencia[] cond = new Residencia[5];

        for (int i=0; i<5; i++) {
            CasaQuad c = new CasaQuad(
                                Math.random()*100);
            Residencia r = new Residencia(c,null);
            cond[i] = r;
        }

        for (Residencia r : cond)
            System.out.println(r.area());
        System.out.println();

        Comparador c = new ComparaArea();
        Ordenador ord = new Selecao();
        ord.ordena(cond,c);

        for (Residencia r : cond)
            System.out.println(r.area());
    }
}
```

Interfaces

- E como fica a interface propriamente dita?

```
public interface Ordenador {
    void ordena(Residencia[] v,
                        Comparador c);
}
```

- E o código que faz a chamada?
  - Ficou mais geral, por permitir que o ordenador seja passado por parâmetro, por exemplo.

```
class X {
    public static void main(String[] args) {
        Residencia[] cond = new Residencia[5];

        for (int i=0; i<5; i++) {
            CasaQuad c = new CasaQuad(
                                Math.random()*100);
            Residencia r = new Residencia(c,null);
            cond[i] = r;
        }

        for (Residencia r : cond)
            System.out.println(r.area());
        System.out.println();

        Comparador c = new ComparaArea();
        Ordenador ord = new Selecao();
        ord.ordena(cond,c);

        for (Residencia r : cond)
            System.out.println(r.area());
    }
}
```

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução
  - Facilita o desenvolvimento de sistemas grandes, envolvendo muitos programadores

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução
  - Facilita o desenvolvimento de sistemas grandes, envolvendo muitos programadores
    - ★ Define-se as interfaces

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução
  - Facilita o desenvolvimento de sistemas grandes, envolvendo muitos programadores
    - ★ Define-se as interfaces
    - ★ Todos as obedecem

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução
  - Facilita o desenvolvimento de sistemas grandes, envolvendo muitos programadores
    - ★ Define-se as interfaces
    - ★ Todos as obedecem
    - ★ Integração posterior mais fácil;

Interfaces – Vantagens

- Se objetos fizerem referência a interfaces e não a classes:
  - Fica mais fácil alterar as classes de um sistema sem ter que alterar aquelas que as utilizam (se a assinatura do método se mantiver igual);
  - Fica fácil implementar polimorfismo de comportamento
    - ★ Classes que mudam de comportamento
    - ★ O chaveamento de comportamento pode ser feito durante compilação ou durante execução
  - Facilita o desenvolvimento de sistemas grandes, envolvendo muitos programadores
    - ★ Define-se as interfaces
    - ★ Todos as obedecem
    - ★ Integração posterior mais fácil;
  - Elimina-se o código repetido.

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

- Além disso, quem implementa a interface pode adicionar métodos extras

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

- Além disso, quem implementa a interface pode adicionar métodos extras
  - Que não necessariamente serão vistos pelo compilador (como na relação subclasse/superclasse)

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

- Além disso, quem implementa a interface pode adicionar métodos extras
  - Que não necessariamente serão vistos pelo compilador (como na relação subclasse/superclasse)
- Interfaces Java também podem definir constantes de classe (atributos static final)

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

- Além disso, quem implementa a interface pode adicionar métodos extras
  - Que não necessariamente serão vistos pelo compilador (como na relação subclasse/superclasse)
- Interfaces Java também podem definir constantes de classe (atributos static final)

```
interface Cores {
    int branco = 0;
    int preto = 255;
}
```

Interfaces – Vantagens

- Em java, objetos podem estender uma única classe
  - Não há herança múltipla
- Contudo, podem implementar quantas interfaces quiser:

```
interface A { ... }

interface B { ... }

class C { ... }

class D extends C
      implements A, B { ... }
```

- Além disso, quem implementa a interface pode adicionar métodos extras
  - Que não necessariamente serão vistos pelo compilador (como na relação subclasse/superclasse)
- Interfaces Java também podem definir constantes de classe (atributos static final)
  - Na interface não é preciso colocar o static final

```
interface Cores {
    int branco = 0;
    int preto = 255;
}
```