Aula 22 – Métodos Abstratos e Interfaces

Norton Trevisan Roman

11 de julho de 2018

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

- Esse era o caso de `Residencia`

```
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

- Esse era o caso de `Residencia`
- Bem coxambrado

```
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

- Esse era o caso de `Residencia`
- Bem coxambrado
- Como fazer então?

```
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

- Esse era o caso de `Residencia`
- Bem coxambrado
- Como fazer então?
- Tornar `area` um método abstrato

```
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

```
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

```
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

- 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

- 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

- Quando aplicado a classes, `abstract` faz com que não possam ser instanciadas

```
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

- Quando aplicado a classes, `abstract` faz com que não possam ser instanciadas
  - Não podemos fazer `new Casa()` (nesse exemplo)

```
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`

```
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

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

```
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

- 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

```
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

- 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`

```
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

- E podemos ter parâmetros em métodos abstratos?

```
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

- 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`

```
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

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

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

- Quiser impedir a existência de objetos dela

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

- Quiser impedir a existência de objetos dela
  - Somente as subclasses poderão ter objetos

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

- Quiser impedir a existência de objetos dela
  - Somente as subclasses poderão ter objetos
- Necessitar que algum método seja conhecido na superclasse

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

- Quiser impedir a existência de objetos dela
  - Somente as subclasses poderão ter objetos
- Necessitar que algum método seja conhecido na superclasse
  - Mas que não possa/precise ter código nela

Métodos e Classes Abstratos

Em suma, use classes abstratas quando:

- Quiser impedir a existência de objetos dela
  - Somente as subclasses poderão ter objetos
- Necessitar que algum método seja conhecido na superclasse
  - Mas que não possa/precise ter código nela
  - Apenas faz sentido o código nas subclasses

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;
        }
  }
}

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

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 também poder comparar por valor?

Interfaces

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

Interfaces

- E se quisermos também poder 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 também poder 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

- Além de mudar o método de ordenação (e alguns especificadores em outras classes)

Interfaces

- 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;
        }
  }

  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 esse 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;
        }
  }

  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 esse 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;
        }
  }

  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

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

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

Interfaces

- Como resolver?
- Interfaces

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

Saída

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

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

Interfaces

- Interfaces são como classes contendo apenas as assinaturas de métodos

Interfaces

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

Interfaces

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

Interfaces

- Interfaces são como classes contendo apenas as assinaturas de métodos
  - Não possuem código
  - Todos os métodos implicitamente públicos
    - Não precisamos usar `public`

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

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 e públicos

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 e públicos
  - Por isso, não há `static` em métodos de interfaces – eles não têm um corpo implementado, então não devem poder ser acessados de forma estática

Interfaces

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

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

Interfaces

- Se não 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 não 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 não 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 não 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

- Assim como classes abstratas exigem que suas subclasses implementem seus métodos abstratos, também interfaces exigem que as classes que as implementam implementem seus métodos

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

class Testando implements Teste {



}
```

Interfaces

- Assim como classes abstratas exigem que suas subclasses implementem seus métodos abstratos, também interfaces exigem que as classes que as implementam implementem seus métodos

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

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

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

Interfaces

- As classes 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();
  }
}
```

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?

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
```

```
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?

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
```

- 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();
  }
}
```

Interfaces

- E agora? Vai compilar?

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
```

- 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();
  }
}
```

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

- 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

```
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

- 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

```
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

- 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

- 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

- 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

- Algo assim...

```
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

- Algo assim...

```
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;
        }
  }
}
```

- Então...

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

Interfaces

- Também temos que implementar os comparadores

Interfaces

- Também temos que implementar os comparadores

```
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

- Também temos que implementar os comparadores
- Deverão substituir os métodos `comparaRes` e `comparaResP` de `Residencia`

```
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);
  AreaPiscina p = new AreaPiscina();

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

  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);
  AreaPiscina p = new AreaPiscina();

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

  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

- Sabemos 3 métodos de ordenação

Interfaces

- 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

- 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

- 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

- 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) terá que explicitamente optar por um dos ordenadores

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

  for (int i=0; i<5; i++) {
    CasaQuad c = new CasaQuad(
                 Math.random()*100,1500);
    Residencia r=new Residencia(c,p);
    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) terá que explicitamente optar por um dos ordenadores
- Que fazer?

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

  for (int i=0; i<5; i++) {
    CasaQuad c = new CasaQuad(
                 Math.random()*100,1500);
    Residencia r=new Residencia(c,p);
    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) 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];
  AreaPiscina p = new AreaPiscina();

  for (int i=0; i<5; i++) {
    CasaQuad c = new CasaQuad(
                 Math.random()*100,1500);
    Residencia r=new Residencia(c,p);
    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) 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];
  AreaPiscina p = new AreaPiscina();

  for (int i=0; i<5; i++) {
    CasaQuad c = new CasaQuad(
                 Math.random()*100,1500);
    Residencia r=new Residencia(c,p);
    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;
      }
    }
  }
}
```

Interfaces

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

```
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;
      }
    }
  }
}
```

Interfaces

- Fazemos o mesmo com os demais métodos de ordenação

```
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

- 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];
    AreaPiscina p = new AreaPiscina();

    for (int i=0; i<5; i++) {
      CasaQuad c = new CasaQuad(
                   Math.random()*100,1500);
      Residencia r = new Residencia(c,p);
      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];
    AreaPiscina p = new AreaPiscina();

    for (int i=0; i<5; i++) {
      CasaQuad c = new CasaQuad(
                   Math.random()*100,1500);
      Residencia r = new Residencia(c,p);
      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 referenciarem interfaces e não classes:

Interfaces – Vantagens

Se objetos referenciarem interfaces e não 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 referenciarem interfaces e não 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 referenciarem interfaces e não 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 referenciarem interfaces e não 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 referenciarem interfaces e não classes:

- Facilita-se o desenvolvimento de sistemas grandes, envolvendo muitos programadores

Interfaces – Vantagens

Se objetos referenciarem interfaces e não classes:

- Facilita-se o desenvolvimento de sistemas grandes, envolvendo muitos programadores
  - Definem-se as interfaces

Interfaces – Vantagens

Se objetos referenciarem interfaces e não classes:

- Facilita-se o desenvolvimento de sistemas grandes, envolvendo muitos programadores
  - Definem-se as interfaces
  - Todos as obedecem

Interfaces – Vantagens

Se objetos referenciarem interfaces e não classes:

- Facilita-se o desenvolvimento de sistemas grandes, envolvendo muitos programadores
  - Definem-se as interfaces
  - Todos as obedecem
  - Integração posterior mais fácil;

Interfaces – Vantagens

Se objetos referenciarem interfaces e não classes:

- Facilita-se o desenvolvimento de sistemas grandes, envolvendo muitos programadores
  - Definem-se as interfaces
  - Todos as obedecem
  - Integração posterior mais fácil;
- Elimina-se o código repetido.

Interfaces

- Em java, classes podem estender uma única classe

```
class C { ... }

class D extends C
                    { ... }
```

Interfaces

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

```
class C { ... }

class D extends C
                    { ... }
```

Interfaces

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

```
interface A { ... }

interface B { ... }

class C { ... }

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

Interfaces

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

```
interface A { ... }

interface B { ... }

class C { ... }

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

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

Interfaces

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

```
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

- Interfaces Java também podem definir constantes de classe (atributos `static final`)

Interfaces

- Interfaces Java também podem definir constantes de classe (atributos `static final`)

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

Interfaces

- Interfaces Java também podem definir constantes de classe (atributos `static final`)
  - Na interface não é preciso colocar o `static final`, isso está implícito

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

Videoaula

Não há.