O equals, novamente...

- de acordo com a estratégia anteriormente apresentada, o método equals de uma subclasse deve invocar o método equals da superclasse, para nesse contexto comparar os valores das v.i. lá declaradas.
  - utilização de **super.equals()**

- seja o método equals da classe Aluno (já conhecido de todos)

```
/**
 * Implementação do método de igualdade entre dois Aluno
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */
public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
     Aluno a =  (Aluno) umAluno;
     return(this.nome.equals(a.getNome()) && this.nota == a.getNota()
            && this.numero == a.getNumero());
 }
```

- seja agora o método equals da classe AlunoTE, que é subclasse de Aluno:

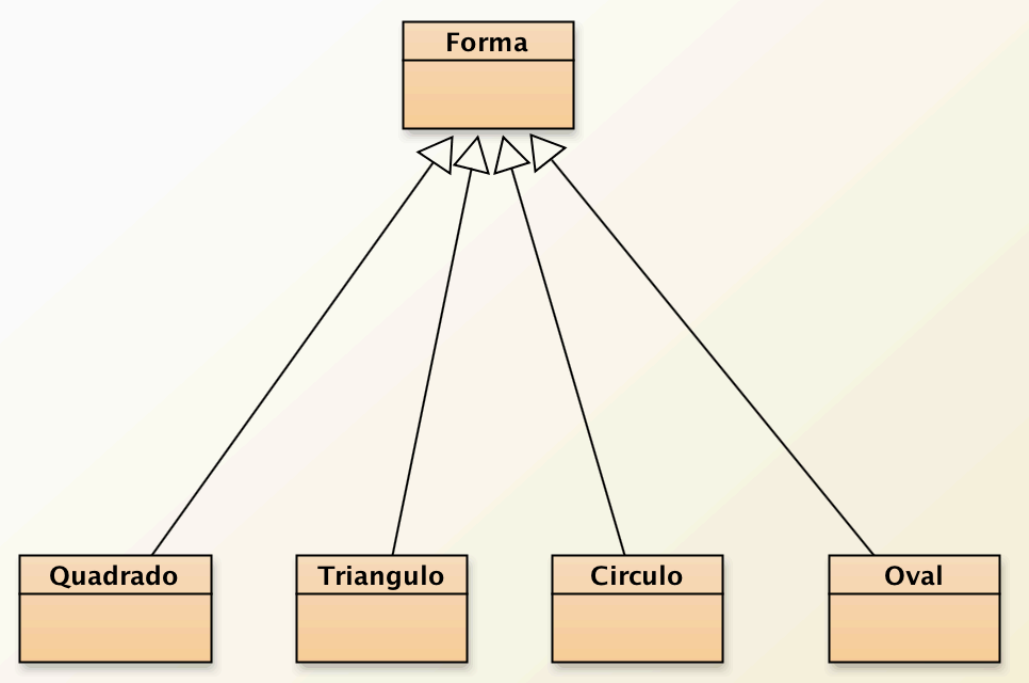
```
/**
 * Implementação do método de igualdade entre dois Alunos do tipo T-E
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */

public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
    AlunoTE a =  (AlunoTE) umAluno;
    return(super.equals(a) & this.nomeEmpresa.equals(a.getNomeEmpresa());
}
```

- considerando o que se sabe sobre os tipos de dados, a invocação **this.getClass()** continua a dar os resultados pretendidos?

Polimorfismo

- capacidade de tratar da mesma forma objectos de tipo diferente
  - desde que sejam compatíveis a nível de API
  - ou seja, desde que exista um tipo de dados que os inclua

Hierarquia das Formas Geométricas

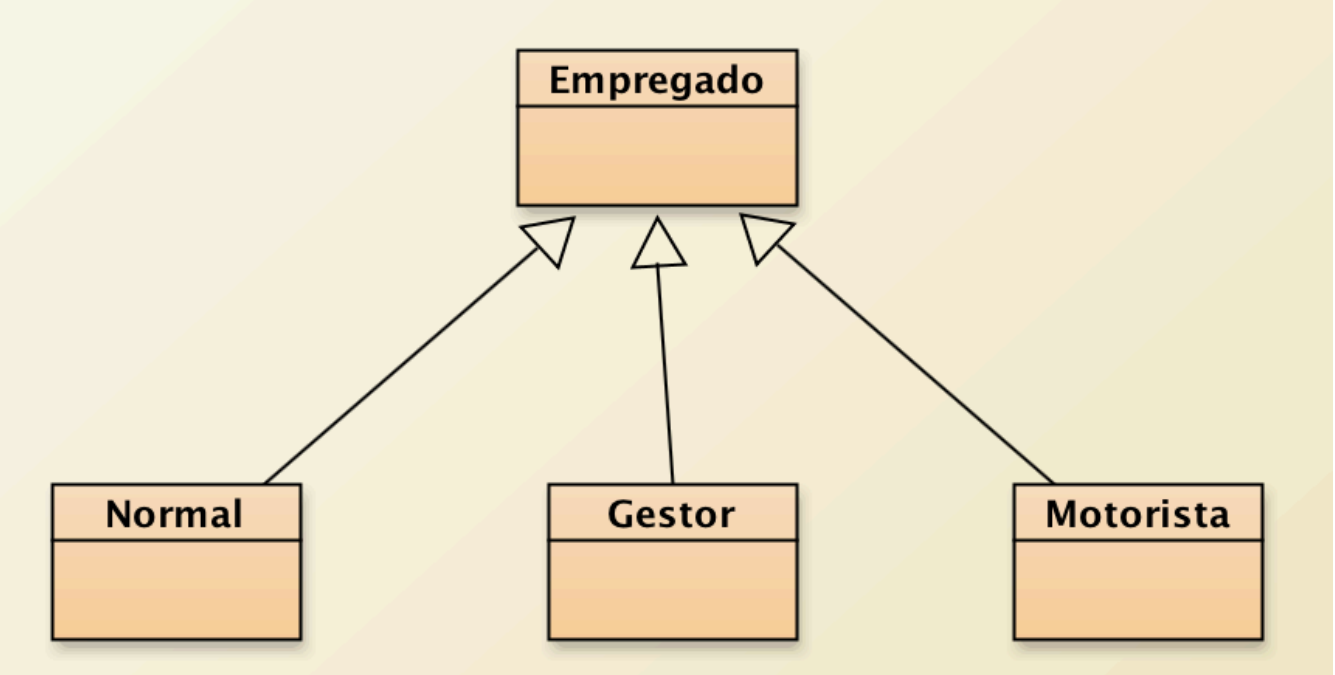


- todas as formas respondem a area() e a perimetro()

- sendo assim é possível tratar de forma igual as diversas instâncias de Forma

```
public double totalArea() {
  double total = 0.0;
  for (Forma f: this.formas)
    total += f.area();
  return total;
}

public int qtsCirculos() {
  int total = 0;
  for (Forma f: this.formas)
    if (f instanceof Circulo) total++;
  return total;
}

public int qtsDeTipo(String tipo) {
  int total = 0;
  for (Forma f: this.formas)
    if ((f.getClass().getSimpleName()).equals(tipo))
      total++;
  return total;
}
```

- Projecto dos Empregados



- os diferentes empregados tem uma definição distinta do método salario()

- Em Empregado Normal:

```
public double salario() {
  return this.getDias()*getSalDia();
}
```

- Em Gestor:

```
public double salario() {
  return this.getDias()*getSalDia()*this.premio;
}
```

- Em Motorista:

```
public double salario() {
   return this.getDias()*getSalDia() + getValorKm()*this.nKms;
 }
```

- os gestores tem um prémio de produtividade e os motoristas recebem em função dos kms.

```
public double totalSalarios() {
    double total = 0.0;
    for(Empregado emp : emps) total += emp.salario();
    return total;
}

public int totalGestores() {
    int total = 0;
    for(Empregado emp : emps) if(emp instanceof Gestor) total++;
    return total;
}

public int totalDe(String Tipo) {
   int total = 0;
    for(Empregado emp : emps)
      if(emp.getClass().getName().equals(Tipo)) total++;
    return total;
}

public double totalKms() {
    double totalKm = 0.0;
    for(Empregado emp : emps)
      if(emp instanceof Motorista) totalKm += ((Motorista) emp).getKms();
    return totalKm;
}
```

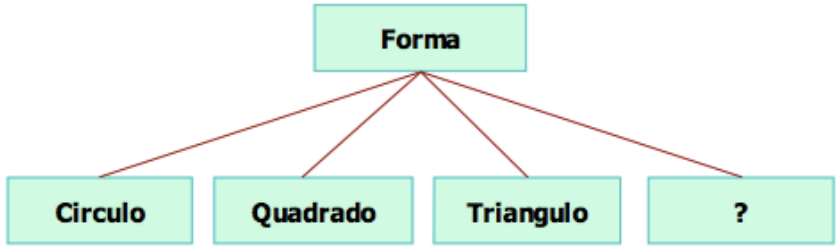
- todos respondem a salario(), embora com implementações diferentes

Classes Abstractas

- até ao momento todas as classes definiram completamente todo o seu estado e comportamento
- no entanto, na concepção de soluções por vezes temos situações em que o código de uma classe pode não estar completamente definido
  - esta é uma situação comum em POO e podemos tirar partido dela para criar soluções mais interessantes

- consideremos que precisamos de manipular forma geométricas (triângulos, quadrados e círculos)
  - no entanto podemos acrescentar, com o evoluir da solução, mais formas geométricas
  - torna-se necessário uniformizar a API que estas classes tem de respeitar
    - todos tem de ter **`area()`** e **`perimetro()`**

- Seja então a seguinte hierarquia:



  - conceptualmente correcta e com capacidade de extensão através da inclusão de novas subclasses de forma

  - mas qual é o estado e comportamento de Forma?

- A classe Forma pode definir algumas v.i., como um ponto central (um Ponto2D), mas se quiser definir os métodos area() e perímetro() como é que pode fazer?
  - Solução 1: não os definir deixando isso para as subclasses
    - as subclasses podem nunca definir estes métodos e aí perde-se a capacidade de dizer que todas as formas respondem a esses métodos

- Solução 2: definir os métodos area() e perimetro() com um resultado inútil, para que sejam herdados e redefinidos
- Solução 3: aceitar que nada pode ser escrito que possa ser aproveitado pelas subclasses e que a única declaração que interessa é a assinatura do método a implementar
    - a maioria das linguagens por objectos aceitam que as definições possam ser incompletas

- em POO designam-se por **classes abstractas** as classes nas quais, pelo menos, um método de instância não se encontra implementado, mas apenas declarado
    - são designados por **métodos abstractos** ou virtuais
    - uma classe 100% abstracta tem apenas assinaturas de métodos

- no caso da classe Forma não faz sentido definir os métodos area() e perímetro, pelo que escrevemos apenas:

```
public abstract class Forma {
//
   public abstract double area();
   public abstract double perimetro();
}
```

- como os métodos não estão definidos, não é possível criar instâncias de classes abstractas

- apesar de ser uma classe abstracta, o mecanismo de herança mantém-se e dessa forma uma classe abstracta é também um (novo) tipo de dados
    - compatível com as instâncias das suas subclasses
    - torna válido que se faça
      `Forma f = new Triangulo()`

- uma classe abstracta ao não implementar determinados métodos, **obriga** a que as suas subclasses os implementem
  - se não o fizerem, ficam como abstractas
- para que servem métodos abstractos?
  - para garantir que as subclasses respondem àquelas mensagens de acordo com a implementação desejada

- Em resumo, as classes abstractas são um mecanismo muito importante em POO, dado que permitem:
  - escrever especificações sintácticas para as quais são possíveis múltiplas implementações
  - fazer com que futuras subclasses decidam como querem implementar esses métodos

- Classe Circulo

```
public class Circulo extends Forma {
   // variáveis de instância
   private double raio;
   // construtores
   public Circulo() {  raio = 1.0; }
   public Circulo(double r) {  raio = (r <= 0.0 ? 1.0 : r); }
   // métodos de instância
   public double area() { return PI*raio*raio; }
   public double perimetro() { return 2*PI*raio; }
   public double raio() {  return raio; }
}
```

- Classe Rectangulo

```
public class Rectangulo extends Forma {
   // variáveis de instância
   private double comp, larg;
   // construtores
   public Rectangulo() {  comp = 0.0; larg = 0.0; }
   public Rectangulo(double c, double l) {  comp = c; larg = l; }
   // métodos de instância
   public double area() { return comp*larg; }
   public double perimetro() { return 2*(comp+larg); }
   public double largura() {  return larg; }
   public double comp() {  return comp; }
 }
```

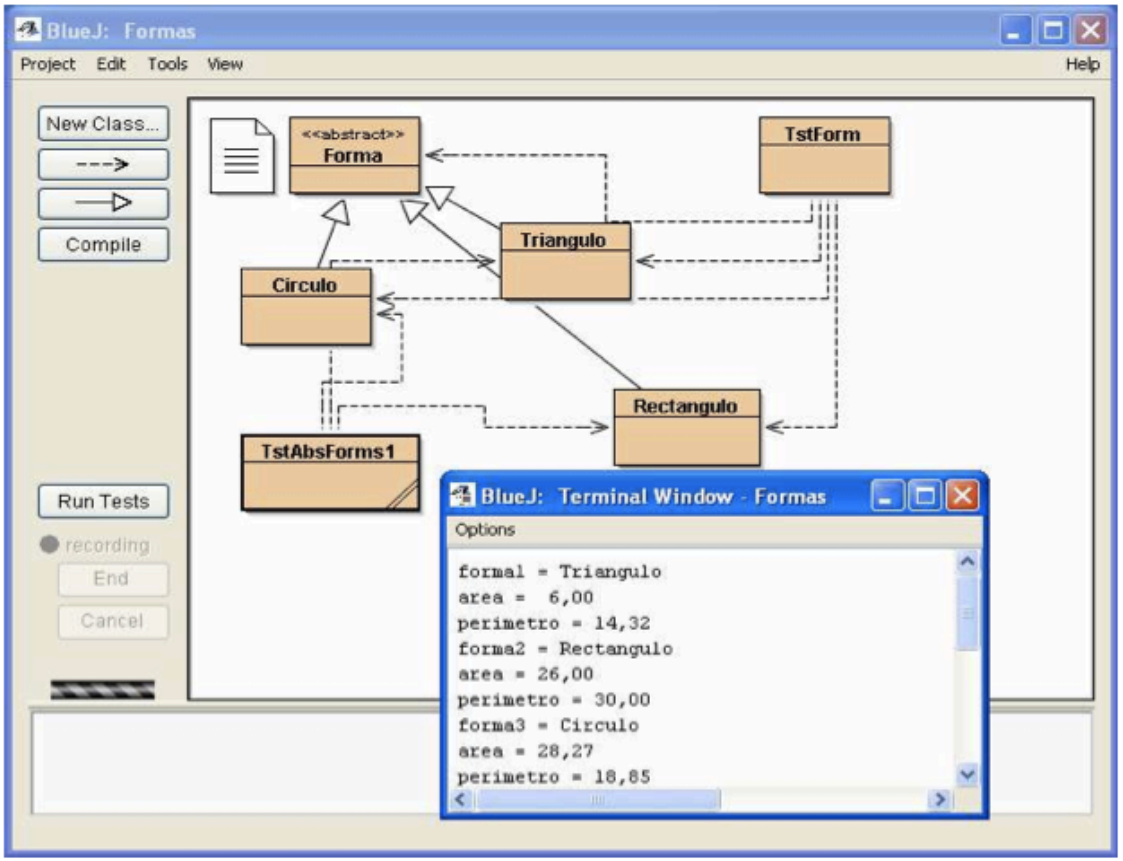
- Classe Triangulo:

```
public class Triangulo extends Forma {
   /* altura tirada a meio da base */
   // variáveis de instância
   private double base, altura;
   // construtores
   public Triangulo() { base = 0.0; altura = 0.0; }
   public Triangulo(double b, double a) {
      base = b; altura = a;
   }
   // métodos de instância
   public double area() { return base*altura/2; }
   public double perimetro() {
      return base + (2*this.hipotenusa()); }
   public double base() { return base; }
   public double altura() { return altura; }
   public double hipotenusa() {
      return sqrt(pow(base/2, 2.0) + pow(altura, 2.0)) ;
   }
}
```

- Classe Forma (com mais informação):

```
public abstract class Forma {

    private Ponto2D ref;

    public Forma() {
        this.ref = new Ponto2D(0,0);
    }

    public Ponto2D getRef() {
      return ref.clone();
    }

    public String dizOla() {
        return "Olá eu sou uma Forma no ponto " + this.getRef() ;
    }

    public abstract double area();
    public abstract double perimetro();
    public abstract String toString();
    public abstract Forma clone();

}
```

- execução do envio dos métodos a diferentes objectos - respostas diferentes consoante o receptor: **polimorfismo**!!



Herança vs Composição

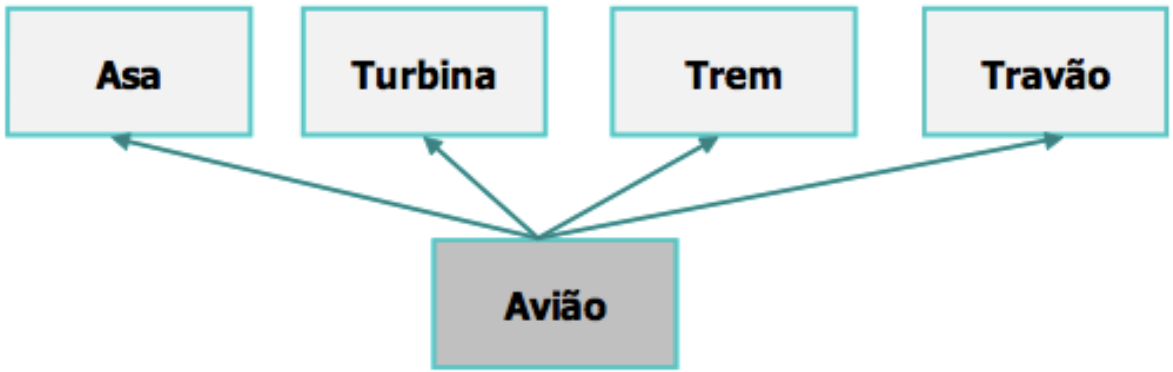
- Herança e composição são duas formas de relacionamento entre classes
  - são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas da mesma forma
- existe uma tendência para se confundir herança com composição

- quando uma classe é criada por composição de outras, isso implica que as instâncias das classes agregadas fazem parte da definição do contentor
  - é uma relação do tipo "parte de" (part-of)
  - qualquer instância da classe vai ser constituída por instâncias das classes agregadas
    - Exemplo: Círculo tem um ponto central (Ponto2D)

- do ponto de vista do ciclo de vida a relação é fácil de estabelecer:
  - quando a instância contentor desaparece, as instâncias agregadas também desaparecem
  - o seu tempo de vida está iminentemente ligado ao tempo de vida da instância de que fazem parte!

- esta é uma forma (e está aqui a confusão) de criar entidades mais complexas a partir de entidades mais simples:
  - Turma é composta por instâncias de Aluno
  - Automóvel é composto por Pneu, Motor, Chassis, ...
  - Empresa é composta por instâncias de EmpregadoNormal, Motorista, Gestor, ...

- Por vezes em situação de herança múltipla parece tornar-se apelativa uma solução como:



- embora o que se pretende ter é composição. Na solução apresentada o avião apenas *tem (**é**!)* **uma** asa, **uma** turbina, **um** trem de aterragem e **um** travão.

- no caso de termos herança simples (a que temos em Java) a solução de ter um Avião como subclasse de Asa é (ainda mais) claramente errada.
  - é errado dizer que Avião *is-a* Asa
  - é correcto dizer que Asa *part-of* Avião

- quando uma classe (apesar de poder ter instâncias de outras classes no seu estado interno) for uma especialização de outra, então a relação é de **herança**

- quando não ocorrer esta noção de especialização, então a relação deverá ser de **composição**