

○ 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?

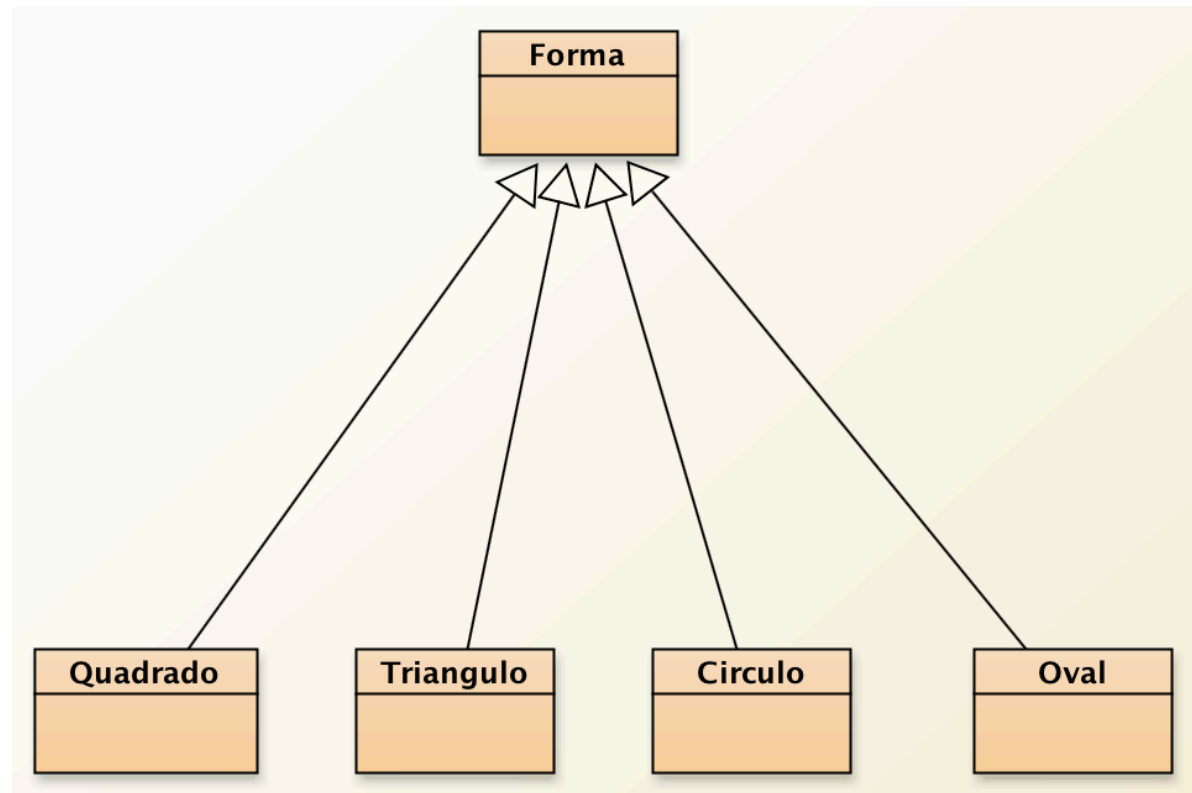
Polimorfismo

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;
}
```

Apesar de termos muitas vantagens em tratar objectos diferentes da mesma forma, por vezes existe a necessidade de saber qual é a natureza de determinado objecto:

determinar qual é a classe de um objecto em tempo de execução

usando `instanceof` ou
`getClass().getSimpleName()`

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

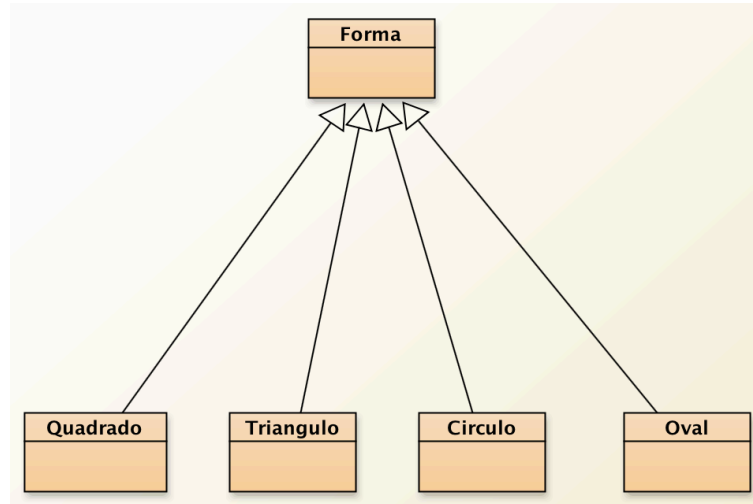
consideremos que precisamos de manipular
forma geométricas (triângulos, quadrados e
círculos)

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 Ponto), a espessura da linha, etc., mas se quisermos definir os métodos `area()` e `perímetro()` como é que podemos fazer?

Solução I: 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 perimetro(), pelo que escrevemos apenas:

```
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 sintáticas para as quais são possíveis múltiplas implementações

fazer com que futuras subclasses decidam como querem implementar esses métodos

Na classe Circulo temos:

```
public double area() {  
    return Math.PI * Math.pow(this.raio,2);  
}
```

```
public double perimetro() {  
    return 2 * Math.PI * this.raio;  
}
```

e em Rectangulo:

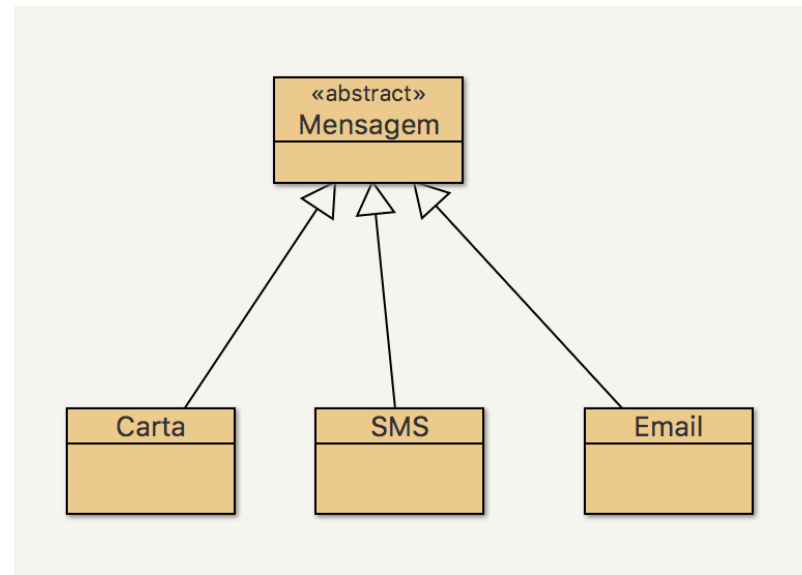
```
public double area() {  
    return this.ladoL * this.ladoA;  
}
```

```
public double perimetro() {  
    return 2 * this.ladoL + 2 * this.ladoA;  
}
```

Podemos aproveitar a capacidade que os métodos abstractos proporcionam para impor comportamento às subclasses para ser mais extensivo na sua identificação:

```
public abstract double area();  
public abstract double perimetro();  
public abstract String toString();  
public abstract FiguraGeometrica clone();
```

Seja a seguinte hierarquia:



cada uma das classes representa uma forma de mensagem. O que é comum a todas é a existência de uma variável “texto”

```
public abstract class Mensagem {  
    private String texto;  
  
    public Mensagem() {  
        this.texto = "";  
    }  
  
    public Mensagem(String texto) {  
        this.texto = texto;  
    }  
  
    public abstract String processa();  
  
    public String getTexto() {  
        return this.texto;  
    }  
  
    public void setTexto(String texto) {  
        this.texto = texto;  
    }  
}
```

```
public class Carta extends Mensagem {  
    private String enderecoOrigem;  
    private String enderecoDestino;  
  
    public Carta() {  
        super();  
        this.enderecoOrigem = "";  
        this.enderecoDestino = "";  
    }  
  
    public Carta(String remetente, String destinatario, String texto) {  
        super(texto);  
        this.enderecoOrigem = remetente;  
        this.enderecoDestino = destinatario;  
    }  
  
    public String processa() {  
        return "CARTA: Destinatário: " + this.enderecoOrigem  
            + "\nRemetente: " + "Mensagem: " + this.getTexto();  
    }  
}
```

```

public class SMS extends Mensagem {
    private String numeroOrigem;
    private String numeroDestino;

    public SMS() {
        super();
        this.numeroOrigem = "";
        this.numeroDestino = "";
    }

    public SMS(String nOrig, String nDest, String texto) {
        super(texto);
        this.numeroOrigem = nOrig;
        this.numeroDestino = nDest;
    }

    public String processa() {
        return ""+ this.numeroOrigem + ">> "
            + this.numeroDestino + "SMS: " + this.getTexto();
    }
}

```

```
public class Email extends Mensagem {  
    private String emailOrigem;  
    private String emailDestino;  
    private String assunto;  
    public Email() {  
        super();  
        this.emailOrigem = ""; this.emailDestino = ""; this.assunto = "";  
    }  
    public Email(String emailOrig, String emailDest, String assunto, String texto) {  
        super(texto);  
        this.emailOrigem = emailOrig;  
        this.emailDestino = emailDest;  
        this.assunto = assunto;  
    }  
    public String processa() {  
        return "From :" + this.emailOrigem + "\nTo: " + this.emailDestino  
            + "\nSubject: " + this.assunto + "\nTexto: " + this.getTexto();  
    }  
}
```



```
public class SistemaMensagens {  
    private List<Mensagem> mensagens;  
  
    public SistemaMensagens() {  
        this.mensagens = new ArrayList<>();  
    }  
  
    // ...  
  
    public int qtsEmails() {  
        return (int) this.mensagens.stream().filter(m -> m instanceof Email).count();  
    }  
  
    public int qtsDeTipo(String tipo) {  
        return (int) this.mensagens.stream().  
            filter(m -> m.getClass().getSimpleName().equals(tipo)).count();  
    }  
}
```

o método `todasAsMensagens()` invoca o método polimórfico `processa()`, que é implementado de forma diferente em todas as classes

```
public String todasAsMensagens() {  
    StringBuilder sb = new StringBuilder();  
    sb.append("Todas as mensagens a enviar:\n");  
    for (Mensagem m: this.mensagens)  
        sb.append(m.processa()+"\n");  
  
    return sb.toString();  
}
```

Classe de teste:

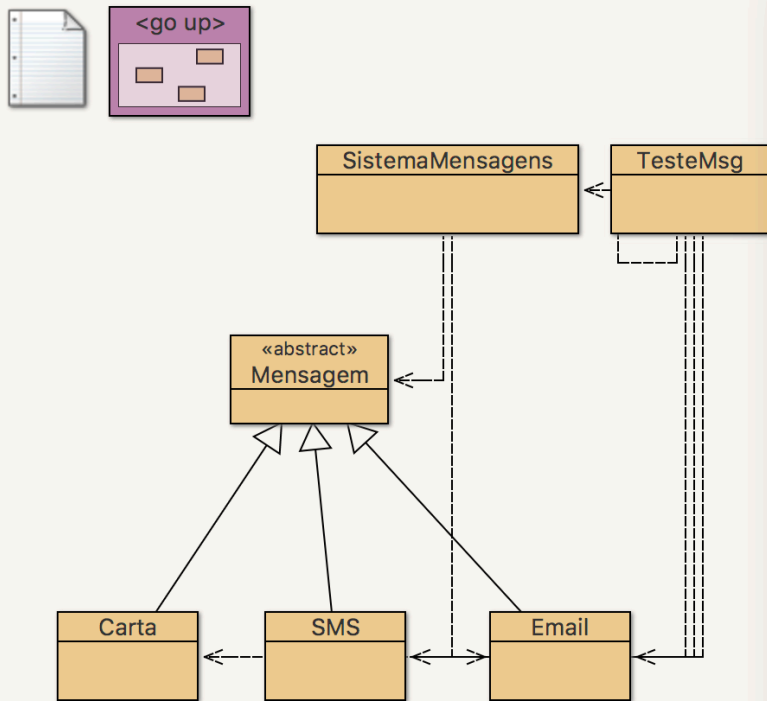
```
public static void main(String[] args) {
    SistemaMensagens sm = new SistemaMensagens();

    Carta c1 = new Carta("José Francisco", "Pedro Xavier", "Em anexo a proposta de compra.");
    Carta c2 = new Carta("Produtos Estrela", "Joana Silva", "Junto enviamos factura.");
    SMS s1 = new SMS("961234432", "929745228", "Estou à espera!");
    SMS s2 = new SMS("911254535", "939541928", "Hoje não há aula...");
    Email e1 = new Email("anr", "jfc", "Teste P00", "Junto envio o enunciado.");
    Email e2 = new Email("a77721", "a55212", "Apontamentos", "Onde estão as fotocópias?");
    Email e3 = new Email("anr", "a43298", "Re: Entrega Projecto", "Recebido.");

    sm.addMensagem(c1); sm.addMensagem(c2);
    sm.addMensagem(s1); sm.addMensagem(s2);
    sm.addMensagem(e1); sm.addMensagem(e2); sm.addMensagem(e3);

    System.out.println("Número de Emails: " + sm.qtsEmails());
    System.out.println("Número de SMS: " + sm.qtsDeTipo("SMS"));

    System.out.println(sm.todasAsMensagens());
}
```



Número de Emails: 3

Número de SMS: 2

Todas as mensagens a enviar:

CARTA: Destinatário: José Francisco

Remetente: MENSAGEM: Em anexo a proposta de compra.

CARTA: Destinatário: Produtos Estrela

Remetente: MENSAGEM: Junto enviamos factura.

961234432>> 929745228SMS: Estou à espera!

911254535>> 939541928SMS: Hoje não há aula...

From :anr

To: jfc

Subject: Teste P00

Texto: Junto envio o enunciado.

From :a77721

To: a55212

Subject: Apontamentos

Texto: Onde estão as fotocópias?

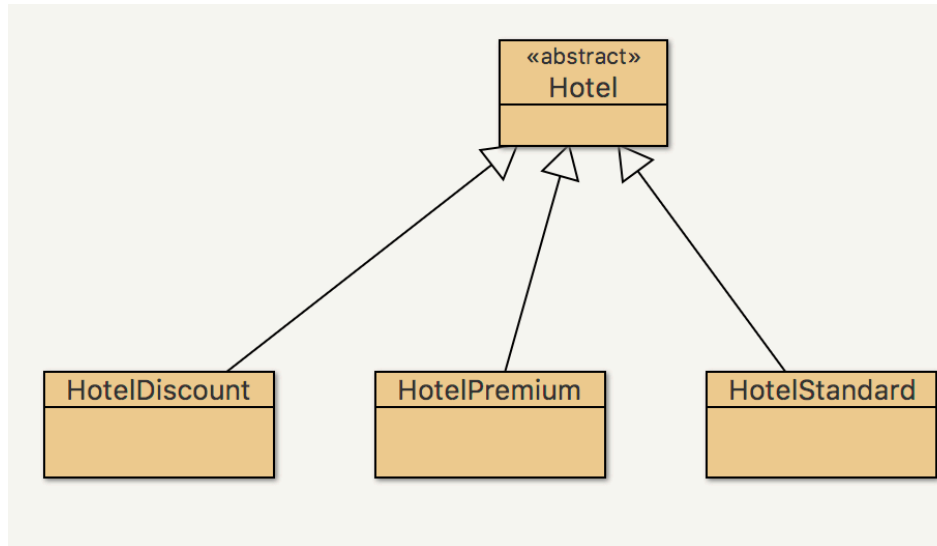
From :anr

To: a43298

Subject: Re: Entrega Projecto

Texto: Recebido.

No caso dos hotéis (Ficha 7)



o método que determina o preço de um quarto é abstracto. A sua concretização é feita em cada uma das subclasses.

HotelStandard:

```
/**
 * Calcula o preço de uma noite no hotel
 * @return valor aumentado da taxa de época alta (se for o caso)
 */

public double precoNoite() {
    return getPrecoBaseQuarto() + (epocaAlta?20:0);
}
```

HotelDiscount:

```
/**
 * Calcula o preço de uma noite no hotel
 * @return valor do preço base afectado pela ocupação.
 */

public double precoNoite() {
    return getPrecoBaseQuarto() * 0.75 + getPrecoBaseQuarto() * 0.25 * ocupacao;
}
```