Modificadores e redefinição de métodos

- a possibilidade de redefinição de métodos está condicionada pelo tipo de modificadores de acesso do método da superclasse (private, public, protected, package) e do método redefinidor
- o método redefinidor não pode diminuir o nível de acessibilidade do método redefinido

- os métodos public podem ser redefinidos por métodos public
- métodos protected por public ou protected
- métodos package por public ou protected ou package

Compatibilidade entre classes e subclasses

- uma das vantagens da construção de uma hierarquia é a reutilização de código, mas...
- os aspectos relacionados com a criação de tipos de dados são também não negligenciáveis
- as classes são associadas estaticamente a tipos
  - uma classe é um tipo de dados

- é preciso saber qual a compatibilidade entre os tipos das diferentes classes (superclasses e subclasses)

  - a questão determinante é saber se uma classe é compatível com as suas subclasses!

- é importante reter o princípio da substituição de Liskov(*) que diz que...

(*) "Family Values: a behavioral notion of subtyping", Barbara Liskov & Jeanette Wing

- “se uma variável é declarada como sendo de uma dada classe (tipo), é admissível que lhe seja atribuído um valor (instância) dessa classe ou de qualquer das suas subclasses”
  - existe compatibilidade de tipos no sentido ascendente da hierarquia (eixo da generalização)
  - ou seja, uma instância de uma subclasse pode ser atribuída a uma instância da superclasse (`Forma f = new Triangulo()`)

- seja o código

```
ClasseA a1, a2;

a1 = new ClasseA();
a2 = new ClasseB();
```

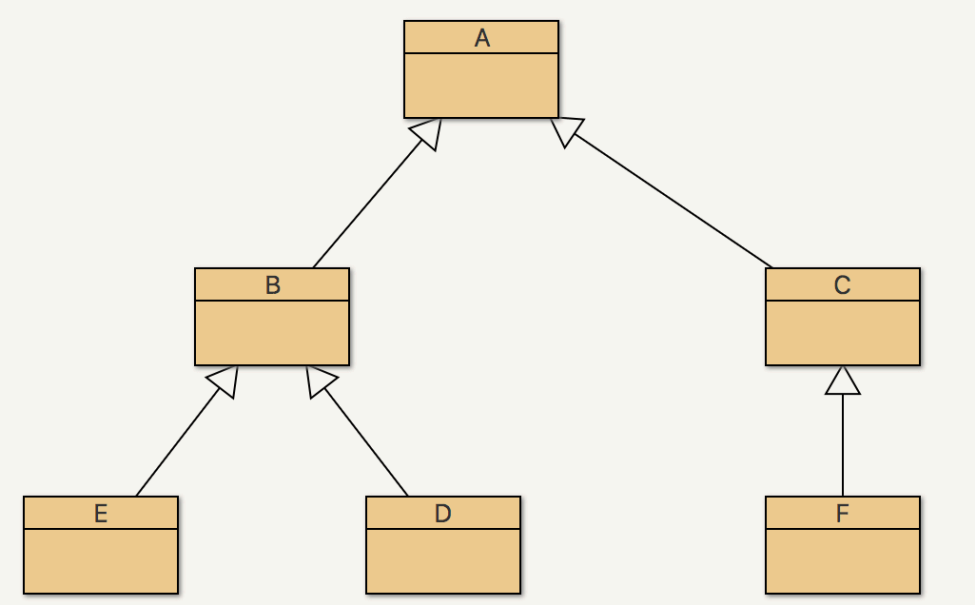
- ambas as declarações estão correctas, tendo em atenção a declaração de variável e a atribuição de valor
  - ClasseB é uma subclasse de ClasseA, pelo que está correcto
- mas o que acontece quando se executa `a2.m()`?

- o compilador tem de verificar se m() existe em ClasseA ou numa sua superclasse (e teria sido herdado)
  - se existir é como se estivesse declarado em ClasseB
  - a expressão é correcta do ponto de vista do compilador
- em tempo de execução terá de ser determinado qual é o método a ser invocado. (cf algoritmo de procura anteriormente apresentado)

- o interpretador, em tempo de execução, faz o ***dynamic binding,*** procurando determinar em função da mensagem qual é o método que deve invocar

- se várias classes da hierarquia implementarem o método m(), então o interpretador executa o método associado ao tipo de dados da **classe do objecto**

- Seja novamente considerada a hierarquia:



- … as implementações das várias classes:

```
public class A {
    private int x;

    public A() {
        this.x = 0;
    }
    public int sampleMethod(int y) {
        return this.x + y;
    }
}
```

```
public class B extends A {
    private int x;

    public B() {
        this.x = 10;
    }
    public int sampleMethod(int y) {
        return this.x + 2* y;
    }
}
```

```
public class C extends A {
    private int x;

    public C() {
        this.x = 20;
    }
    public int sampleMethod(int y) {
        return this.x + 2*y;
    }
}
```

```
public class E extends B {
    private int x;

    public E() {
        this.x = 100;
    }
    public int sampleMethod(int y) {
        return this.x + 10*y;
    }
}
```

```
public class D extends B {
    private int x;

    public D() {
        this.x = 100;
    }
    public int sampleMethod(int y) {
        return this.x + 20*y;
    }
}
```

```
public class F extends C {
    private int x;

    public F() {
        this.x = 200;
    }
    public int sampleMethod(int y) {
        return this.x + 3*y;
    }
}
```

- do ponto de vista dos tipos de dados especificados e da relação entre eles, podemos estabelecer as seguintes relações:
  - um B é um A, um C é um A
  - um E é um B, um D é um B
  - um F é um C
  - ou seja, um D pode ser visto como um B ou um A. Um F pode ser visto como um A, etc...

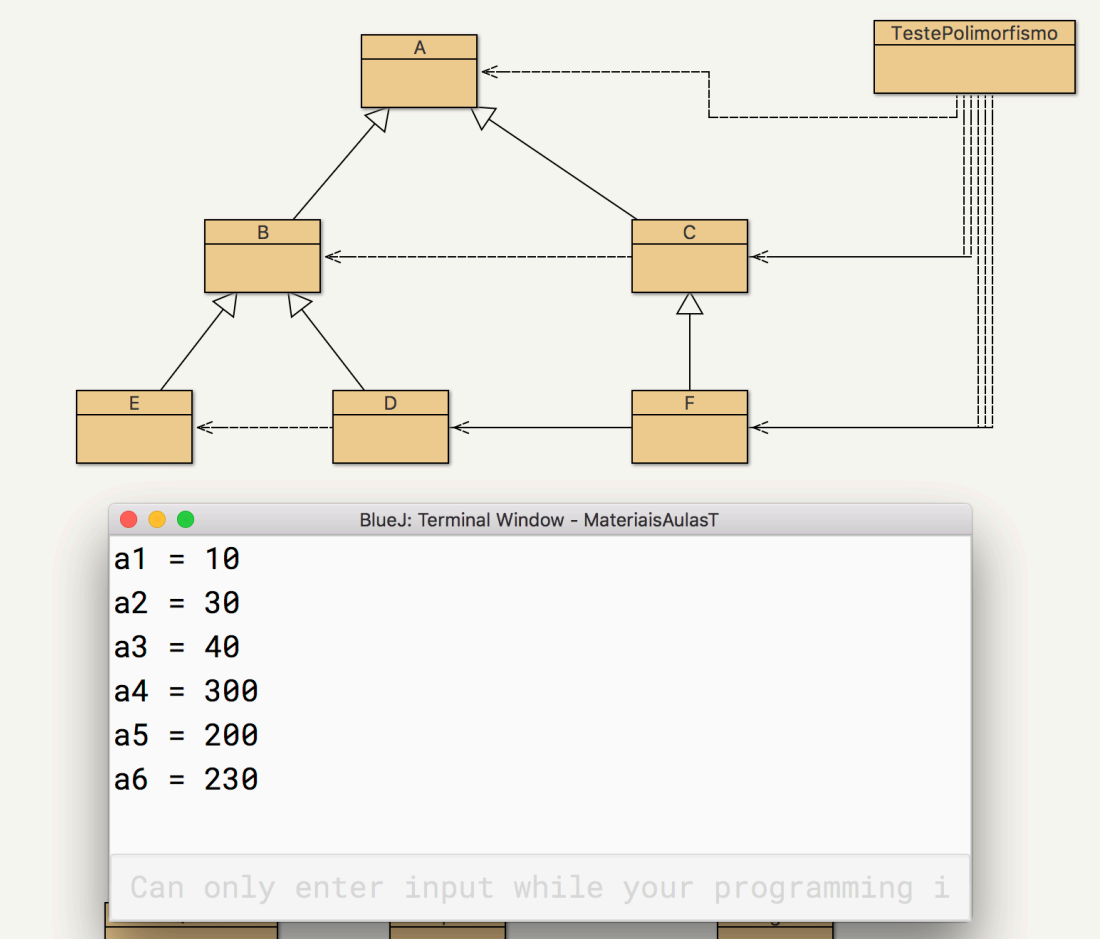
- considere-se o seguinte programa teste:

```
public static void main(String[] args) {

  A a1,a2,a3,a4,a5,a6;
  a1 = new A();
  a2 = new B();
  a3 = new C();
  a4 = new D();
  a5 = new E();
  a6 = new F();

  System.out.println("a1 = " + a1.sampleMethod(10));
  System.out.println("a2 = " + a2.sampleMethod(10));
  System.out.println("a3 = " + a3.sampleMethod(10));
  System.out.println("a4 = " + a4.sampleMethod(10));
  System.out.println("a5 = " + a5.sampleMethod(10));
  System.out.println("a6 = " + a6.sampleMethod(10));
}
```

- qual é o resultado?

- importa distinguir dois conceitos muito importantes:
  - tipo e*stático* da variável
    - é o tipo de dados da declaração, tal como foi aceite pelo compilador
  - tipo *dinâmico* da variável
    - corresponde ao tipo de dados associado ao construtor que criou a instância

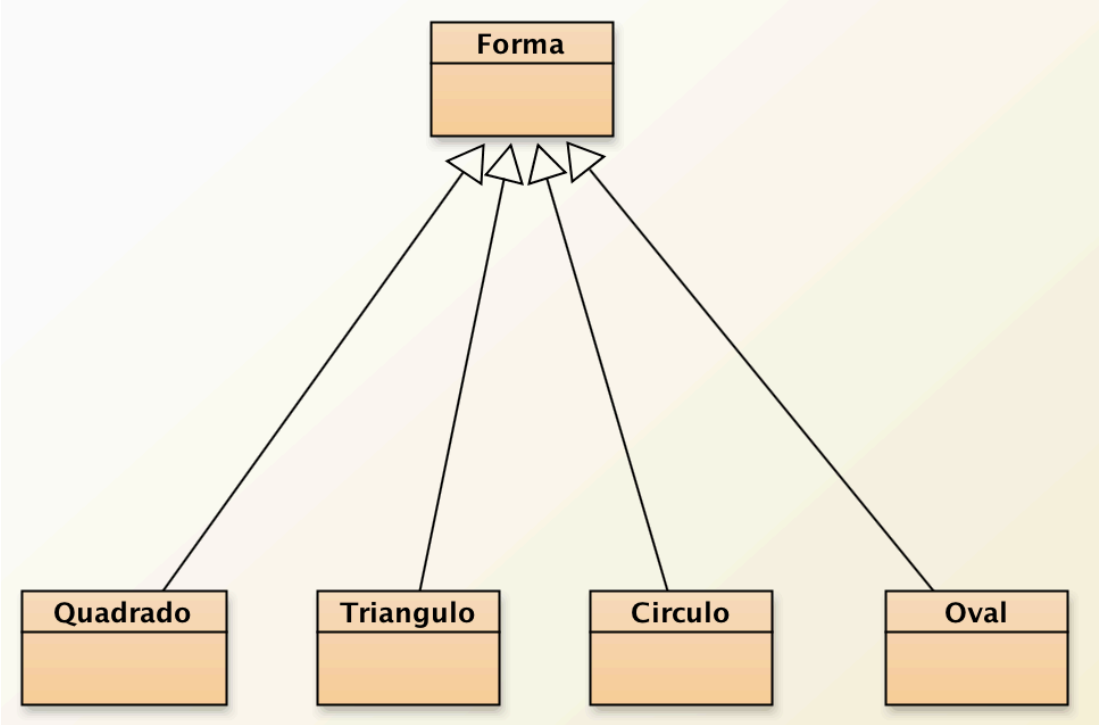
- como o interpretador executa o algoritmo de procura dinâmica de métodos, executando sampleMethod() em cada uma das classes, então o resultado é:



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;
}
```

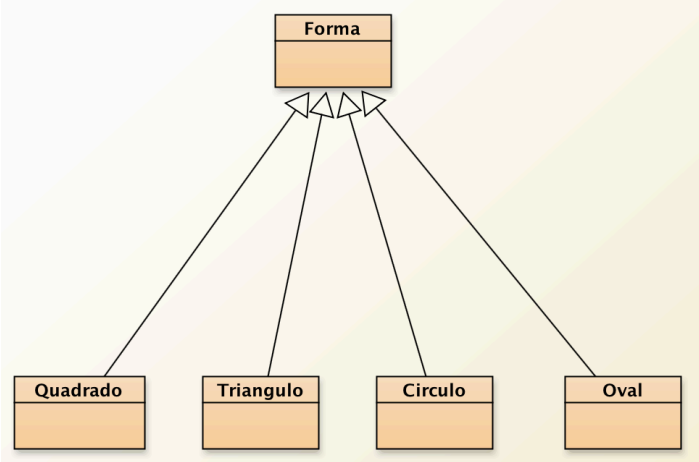
- Apesar de termos muitas vantagens em tratar objectos diferentes da mesma forma, por vezes existe a necessidade de saber qual é a natureza de determinado objecto:
  - determinar qual é a classe de um objecto em tempo de execução
  - usando **`instanceof`** ou **`getClass().getSimpleName()`**

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
    - p. exemplo, todos devem possuir **`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 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 perimetro(), pelo que escrevemos apenas:

```
public abstract double area();
public abstract double perimetro();
```

- como os métodos não estão definidos, a classe será também abstracta e 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

- Na classe Circulo temos:

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

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

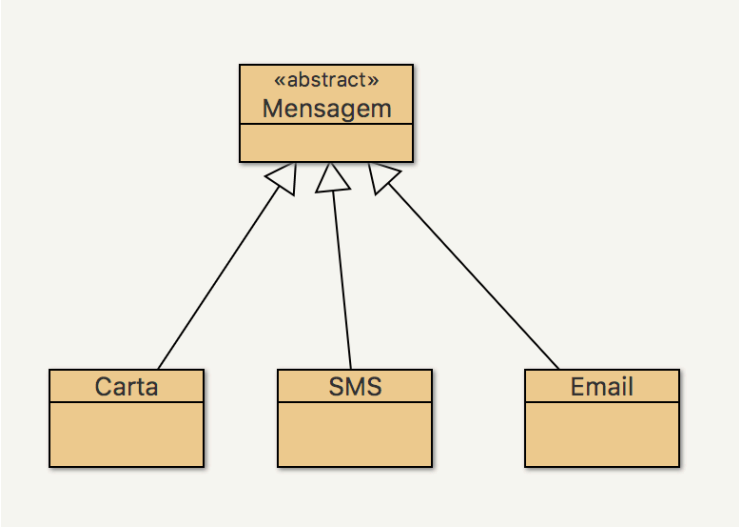
- e em Rectangulo:

```
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:

```
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"

```java
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;
    }
}
```

```java
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();
  }
```

```java
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();
    }
```

```java
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();
  }
```

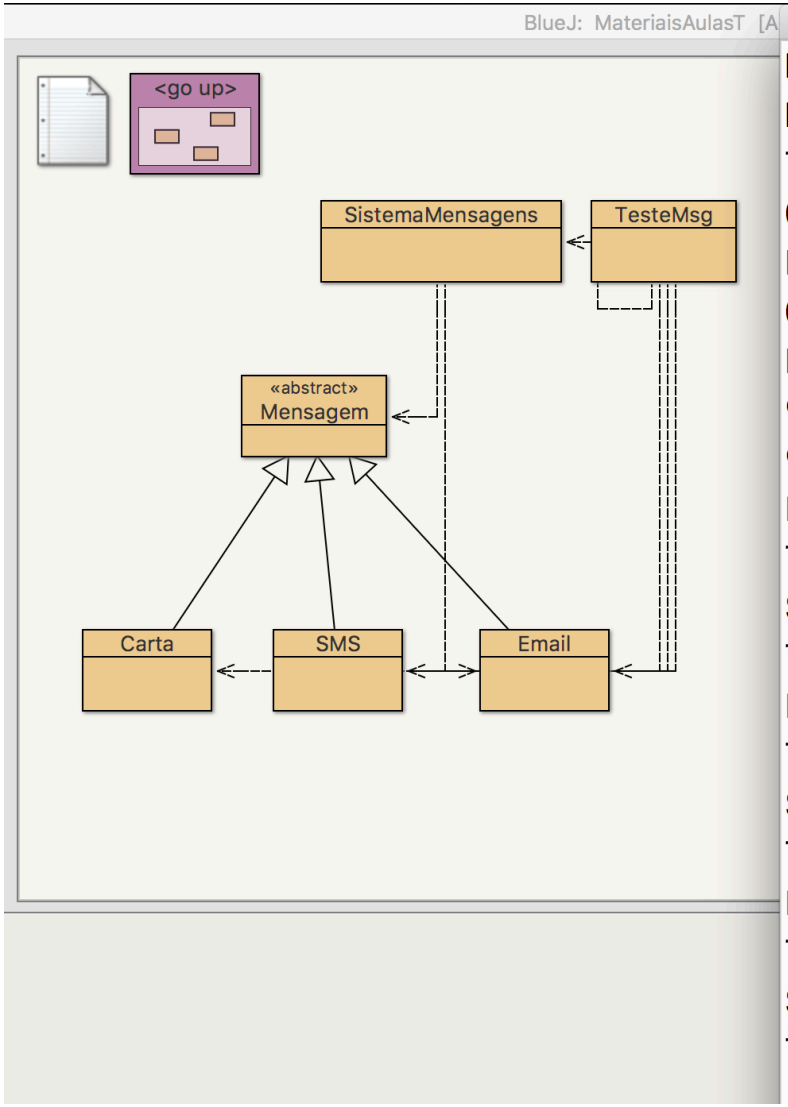
```java
public class SistemaMensagens {
  private List<Mensagem> mensagens;

  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:

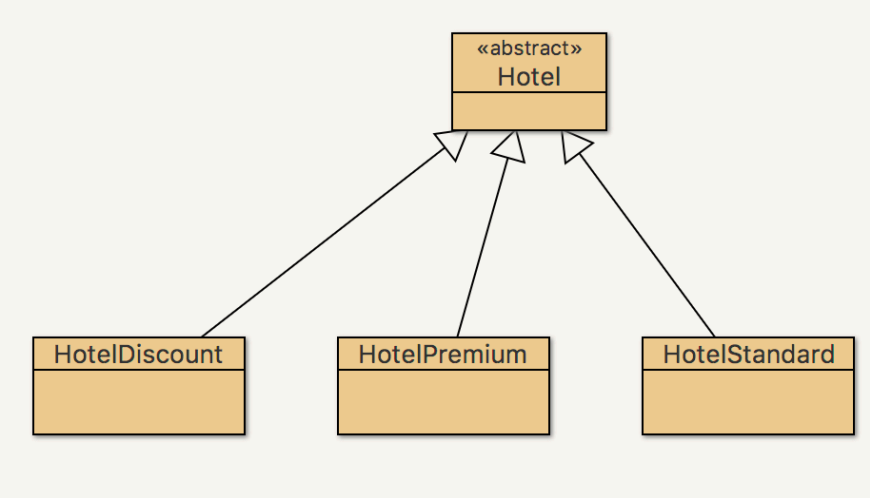
```
public static void main(String[] args) {
  SistemaMensagens sm = new SistemaMensagens();

  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 POO", "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());
```



BlueJ: Terminal Window - MateriaisAulasT

```
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 POO
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.
```

- Num outro exemplo, seja a seguinte hierarquia, em que…



- …o método que determina o preço de um quarto é abstracto. A sua concretização é

- 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;
}
```

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?

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/agregação de outras, isso implica que as instâncias das classes compostas/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 compostas/agregadas
    - Exemplo: Círculo tem um ponto central (Ponto)

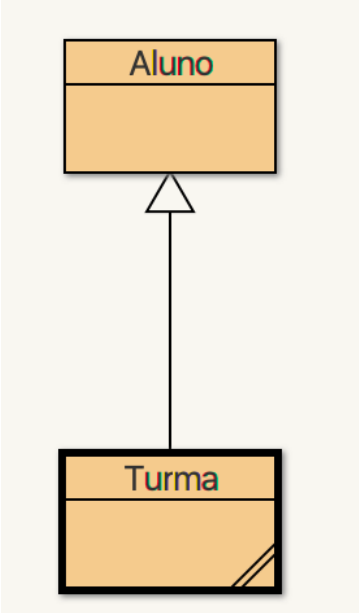
- do ponto de vista do ciclo de vida a relação é fácil de estabelecer:
  - (quando é composição) quando a instância contentor desaparece, as instâncias agregadas também desaparecem
  - o seu tempo de vida está iminentemente ligado ao tempo de vida da instância de que fazem parte!
  - (quando é agregação)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

- 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, …
  - Clube é composto por instâncias de Atleta, Funcionario, Dirigente, ...

- 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/agregação**

- Uma forma simples de testar se faz sentido a relação ser de herança é "ler" o diagrama.
- Por exemplo:



- nunca será possível esta estruturação, pois não é verdade que "uma turma é um aluno"!