

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 (Ponto)

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, ...

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**

Interfaces

Como vimos atrás, as classes além de descreverem estrutura e comportamento, podem ser também vistas como tipos de dados

um tipo de dados representa o comportamento que é possível invocar num objecto desse tipo

O tipo de dados de uma instância é o tipo da classe que a criou (através da invocação do construtor)

Na declaração:

```
ClasseA a = new ClasseA();  
ClasseB b = new ClasseB();
```

os objectos a e b são do tipo de dados que lhes é dado pela classe que os cria.

Os tipos de dados das subclasses podem ser designados como subtipos e já sabemos que:

se ClasseA é superclasse de ClasseB

um objecto do tipo ClasseA pode ser substituído por um objecto do subtipo ClasseB sem alteração nas classes que manipulam objectos do tipo ClasseA.

Este mecanismo de substituição é essencial para as construções polimórficas que estudamos anteriormente.

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

Claro que esta capacidade de substituição de tipos por subtipos só funciona no contexto de uma hierarquia

Esta construção ainda que muito poderosa tem algumas limitações. A saber:

como fazer que classes não relacionadas hierarquicamente possam ser vistas como tendo um tipo comum?

como fazer para esconder, em determinados contextos, comportamento associado a um determinado tipo de dados?

A solução passa por passar a utilizar um outro mecanismo de tipo de dados existente em Java: as **interfaces**.

Interfaces não são classes, mas são apenas declarações sintáticas de expressão de comportamento

não estão na mesma hierarquia das classes e possuem uma hierarquia própria de herança múltipla

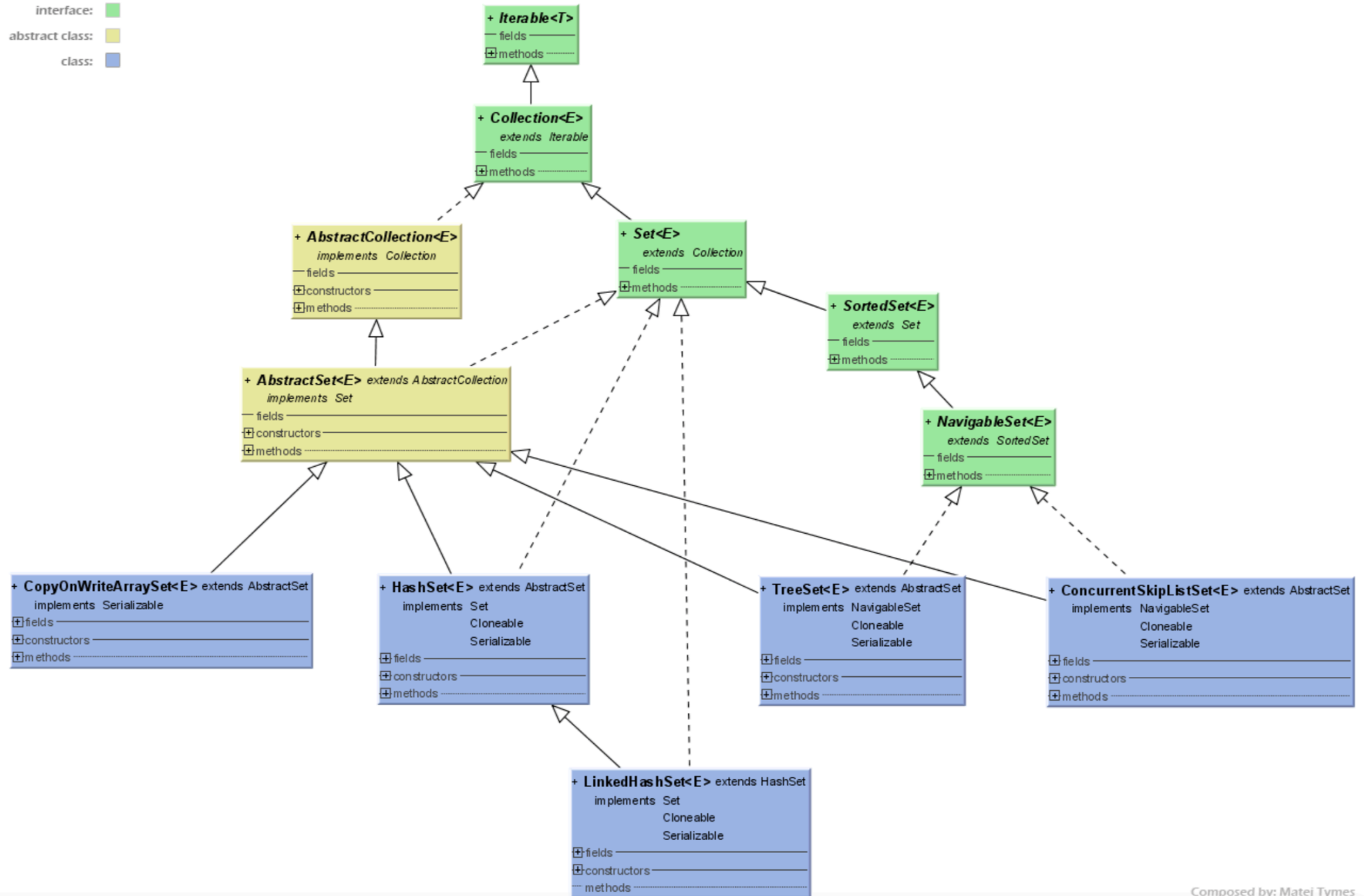
as classes decidem com que tipo de interfaces querem ser compatíveis

Anteriormente já tivemos contacto com interfaces:

Set<T>, List<T>, Map<K, V> não são classes, mas tipos de dados declarados como interfaces

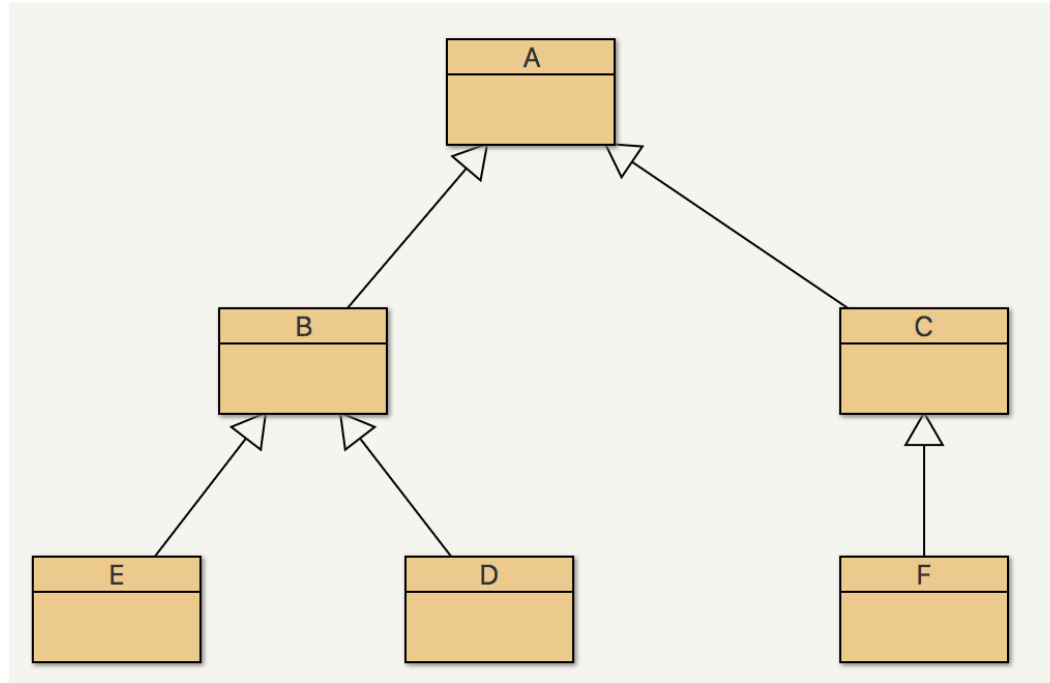
Comparator<T> é uma interface que representa o tipo de dados de todos os objectos que possuem o método `compare (...)` declarado

interface: ■
 abstract class: ■
 class: ■



Composed by: Matej Tymes

Consideremos as seguinte classes:



Se quisermos representar uma colecção de objectos do tipo A (e dos seus subtipos) apenas precisamos de declarar:

`Collection<A>`

Mas o que acontece se quisermos apenas ter numa colecção objectos do tipo D e F?

solução?: aceitamos objectos do tipo A e validamos depois se são D ou F?

solução: criamos um tipo de dados comum só a D e F!

mas como o fazemos na construção hierárquica anterior?

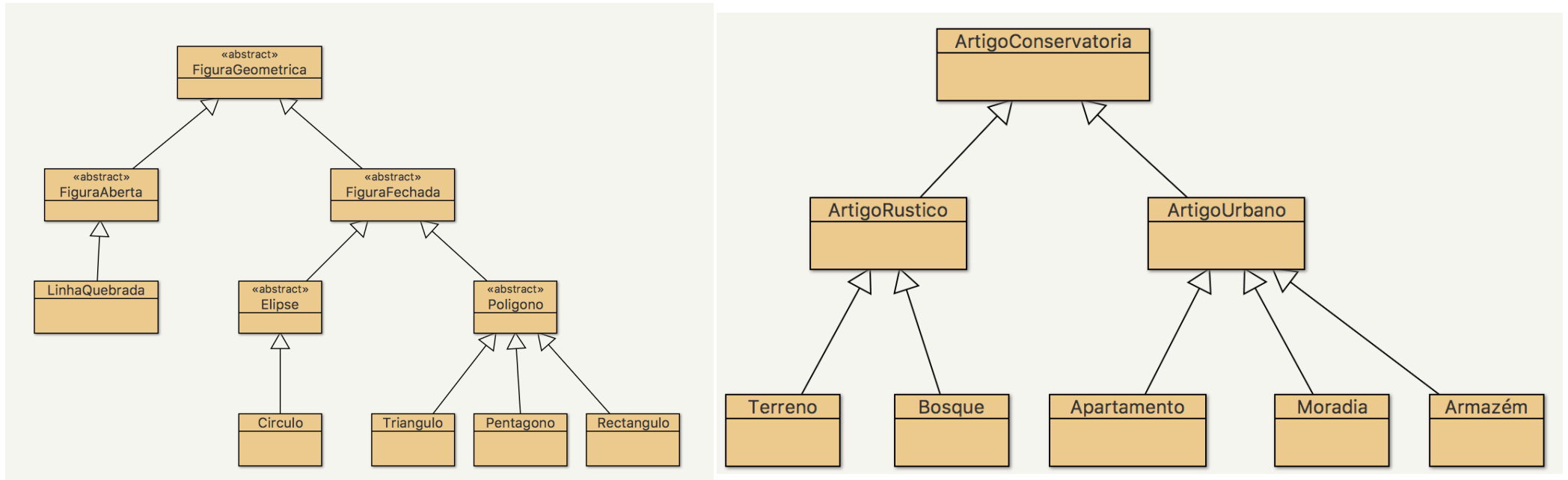
altera-se a hierarquia?

Reescrever a hierarquia quase nunca é uma hipótese viável porque isso tem (muito) impacto nas classes que usam objectos desta hierarquia

é impossível fazer isso de forma silenciosa!

pretendemos ter programação genérica e extensível, mas também estável e sem impactos nas aplicações já existentes.

Outro exemplo:



todas estas classes possuem o método `area()` e `perimetro()`, mas como é que as compatibilizamos por forma a serem tratadas de forma comum?

Criando uma interface, Aravel, que contenha a declaração do comportamento desejado (tipo de dados)

```
public interface Aravel {  
    public double area();  
    public double perimetro();  
}
```

tornando agora possível ter classes que queiram ser compatíveis com este novo tipo de dados.

As classes que queiram ser compatíveis com a interface devem fazer:

```
public class Terreno extends ArtigoRustico implements Aravel {
```

Uma classe pode implementar mais do que uma interface.

```
public class Terreno extends ArtigoRustico implements Aravel, Mensuravel {
```

```
public interface Mensuravel {  
    public double valorMercado();  
}
```

Uma instância de Terreno pode ser vista como sendo:

do tipo de dados Terreno e tem acesso a todos os métodos

do tipo de dados Aravel e tem acesso apenas a area() e perimetro()

do tipo de dados Mensuravel e tem acesso apenas a valorMercado()

Elementos de uma interface

A declaração de interface pode conter:

- um conjunto de métodos abstractos

- um conjunto de constantes

- métodos concretos, designados por *default methods*

- métodos de classe concretos, *static methods*

Hierarquia das Interfaces

As interfaces podem estar colocadas numa estrutura hierárquica, que ao invés da das classes é uma hierarquia múltipla

uma interface pode herdar de mais do que uma interface

nessas situações é preciso ter atenção à herança de métodos com a mesma assinatura

```
NomeInterface.nomeMetodo()
```

Default Methods

Até ao Java 8 as interfaces apenas podiam declarar assinaturas de métodos, que as classes implementadoras deviam tornar concretos.

A partir do Java 8 foi possível acrescentar métodos completamente codificados e que podem ser utilizados pelas classes que implementem a interface.

Consideremos a interface Aravel definida anteriormente. O que acontece que acrescentarmos mais um método abstracto à interface?

as classes que a implementam passam a dar erro de compilação, porque...

falta implementar um método!

Com a utilização dos *default methods* as classes que já implementam a interface não precisam de ser alteradas.

As classes implementadoras que desejam passam a poder utilizar esse método

Se a interface Aravel passar agora a ter um método default:

```
public interface Aravel {  
    public double area();  
    public double perimetro();  
    public default Ponto pontoCentral(...) { <<codigo>> }  
}
```

A classe `Terreno` que implementa `Aravel` tem de implementar os métodos `area()` e `perimetro()` e pode utilizar o método `pontoCentral(...)` já codificado.

As classes que não conhecem a implementação de `pontoCentral(...)` continuam inalteradas e a funcionar devidamente.

Os métodos default permitem adicionar funcionalidade às interfaces já existentes, sem alterar as relações de implementação já existentes

os default methods foram criados para permitirem acrescentar funcionalidade (em Java 8) à API das Collections

Se a classe tiver um método com a mesma assinatura que um default method, prevalece a definição da classe!

Métodos static em interfaces

A partir do Java 8 foi também possível criar métodos de classe na definição das interfaces.

as instâncias das classes que implementam a interface tem acesso a estes métodos

invocáveis da forma

```
Interface.metodo()
```

Classes abstractas e interfaces

A escolha de utilização de uma classe abstracta ou uma interface é por vezes uma decisão difícil.

Uma classe abstracta pode ter variáveis de instância enquanto uma interface não pode.

Uma classe abstracta e uma interface podem declarar métodos abstractos para outras classes implementarem.

No entanto, uma classe abstracta não obriga as subclasses a implementá-los. Se não o fizerem as subclasses ficam como abstractas. Nas interfaces, se não o fizerem, dá erro de compilação.

Uma interface é um tipo de dados (uma API) que pode ser utilizada por qualquer classe. A definição da classe abstracta só é possível dentro da hierarquia.

A partir do Java 8 foi dada uma importância acrescida às interfaces como mecanismo de extensibilidade.

... e ainda

A verificação de tipo pode ser feita da mesma forma que fazemos para as classes, com instanceof

```
List<ArtigoConservatoria> propriedades = new ArrayList<>();  
...  
...  
int numMensuravel = 0;  
for(ArtigoConservatoria ac: this.propriedades)  
    if (ac instanceof Mensuravel)  
        numMensuravel++;
```

na expressão acima não se está a validar a classe, mas sim o tipo de dados estático

Do ponto de vista da concepção de arquitecturas de objectos, as interfaces são importantes para:

- reunirem similaridades comportamentais, entre classes não relacionadas hierarquicamente

- definirem novos tipos de dados

- conterem a API comum a vários objectos, sem indicarem a classe dos mesmos (como no caso do JCF)

Uma classe passa a ter duas vistas (ou classificações) possíveis:

é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento

é subtipo, por se enquadrar numa hierarquia múltipla de definições de comportamento (abstracto ou já implementado cf. default methods)

finalmente...

Classes podem implementar múltiplas interfaces

Em Java8 as interfaces podem:

- incluir métodos **static**

- fornecer implementações por omissão dos métodos (*keyword* **default**)

Functional Interface (Java8)

- uma interface com um único método abstracto (e qualquer número de métodos **default**)

- Instâncias criadas com expressões lambda e com referências a métodos ou construtores

Em resumo...

As interfaces Java são especificações de tipos de dados. Especificam o conjunto de operações a que respondem objectos desse tipo

Uma instância de uma classe é imediatamente compatível com:

- o tipo da classe
- o tipo da interface (se estiver definido)

Enums

Tipo de dados especial que permite variáveis com um conjunto fixo de constantes como valores possíveis

```
public enum DiaDaSemana {  
    SEGUNDA, TERÇA, QUARTA,  
    QUINTA, SEXTA, SÁBADO, DOMINGO  
}
```

```
...  
DiaDaSemana d = DiaDaSemana.SEGUNDA;  
...
```

Enums

```
...  
DiaDaSemana d;  
...  
if (d == DiaDaSemana.DOMINGO)  
    ...  
else  
    ...
```

```
...  
DiadaSemana d;  
...  
switch (d) {  
    case DOMINGO: ...  
    case SEGUNDA: ...  
    ...  
}
```

Nos **switch** os valores dos **case** não são qualificados com o nome do **Enum**.

Enums

Enums são classes pelo que podem ter variáveis de instância e métodos.

Cada constante terá os atributos e métodos.

```
public enum Elemento {  
    ALUMINIO, ANTIMONIO, ARGON, ARSENICO, AZUFRE;  
  
    private double pesoAtomico;  
  
    private double getPesoAtomico() {  
        return pesoAtomico;  
    }  
}
```

Necessário

;!

Mas...

(está incompleto!)
como definir o peso de cada elemento (como inicializar as variáveis de instância)?!

Enums

Podemos inicializar as variáveis de instância na construção das constantes.

Neste caso é necessário definir construtores (tem que ser **package** ou **private**)

```
public enum Elemento {  
    ALUMINIO(26.98), ANTIMONIO(121.8), ARGON(39.95),  
    ARSENICO(74.92), AZUFRE(32.06);  
  
    private double pesoAtomico;  
    Elemento (double p) {  
        this.pesoAtomico = p;  
    }  
  
    public double getPesoAtomico() {  
        return pesoAtomico;  
    }  
}
```