Herança vs Composição

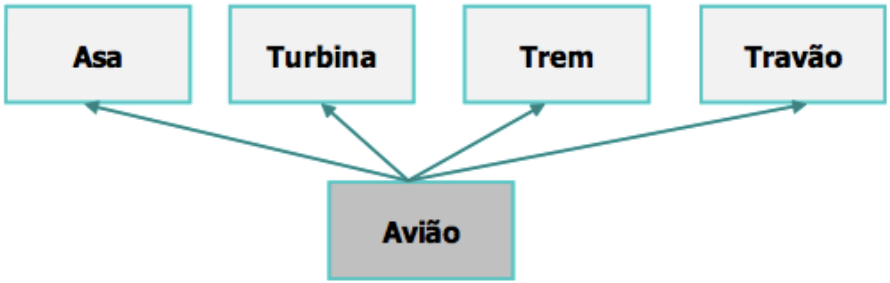
- Herança e composição são duas formas de relacionamento entre classes
  - são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas para o mesmo fim
- existe uma tendência para se confundir herança com composição

- 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 Empregado

- 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 **uma** asa, **uma** turbina, **um** trem de aterragem e **um** travão.

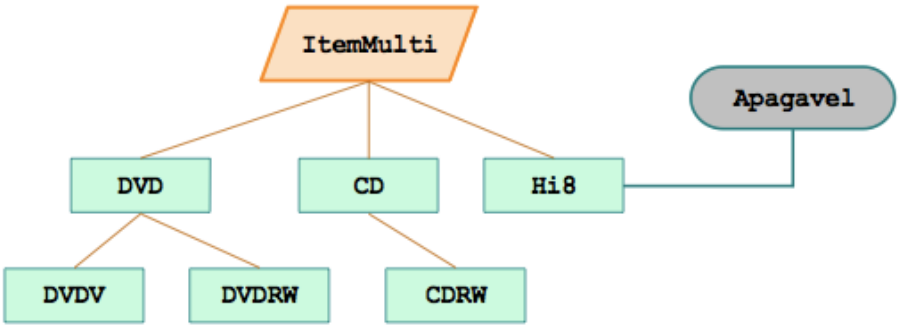
- no caso de termos herança simples (a que temos em Java) a solução de ter um Avião como subclasse de Asa é perfeitamente ridícula.
  - é errado dizer que Asa *is-a* Avião
  - é correcto dizer que Asa *part-of* Avião

- quando uma classe (apesar de 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**

Ainda sobre interfaces...

- Uma hierarquia típica



-

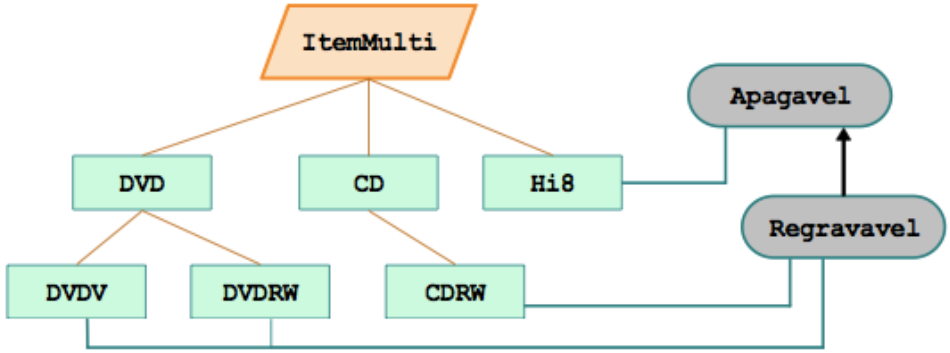
```
public class Hi8 extends ItemMulti implements Apagavel {
   //
   private int minutos;
   private double ocupacao;
   private int gravacoes;
    . . . .
   // implementação de Apagavel
   public void apaga() { ocupacao = 0.0; gravacoes = 0; }
}
```

- Qualquer instância de Hi8 é também do tipo Apagavel, ou seja:

```
Hi8 filme1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);
Apagavel apg1 = filme1;
apg1.apaga();
```

- no entanto, a uma instância de Hi8 que vemos como sendo um Apagavel, apenas lhe poderemos enviar métodos definidos nessa interface (i.e. nesse tipo de dados)

- Temos também a possibilidade de ter vários tipos de dados válidos para diferentes objectos.



- Por vezes, o que provavelmente acontece com Regravavel, a interface é apenas um marcador.

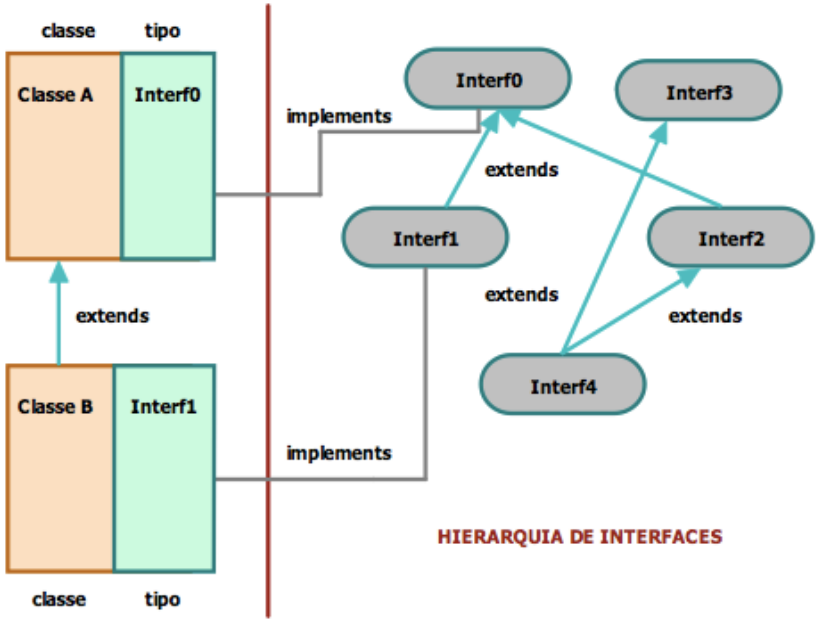
- A verificação de tipo pode ser feita da mesma forma que fazemos para as classes, com instanceof

```
ItemMulti[ ]  filmes = new ItemMulti[ 500] ;
// código para inserção de filmes no array....
int contaReg = 0;
for(ItemMulti filme : filmes)
   if (filme instanceof Regravavel) contaReg++
out.printf("Existem %d items regraváveis.", contaReg);
```

- 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

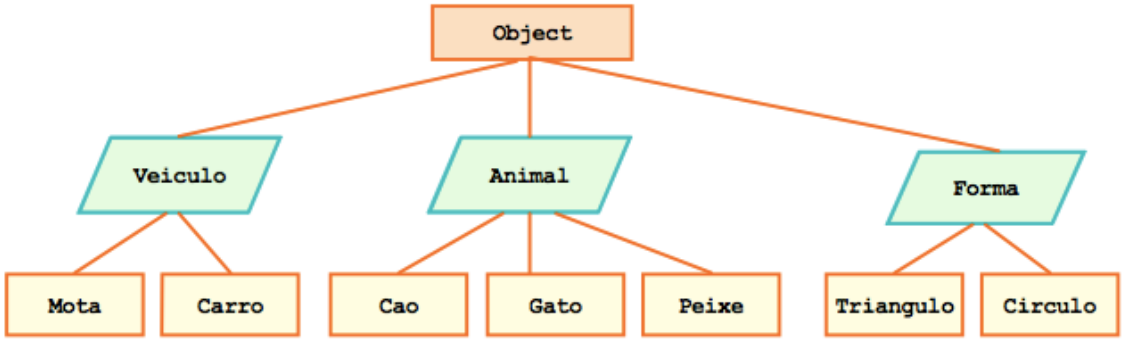
- O modelo geral é assim:



- onde coexistem as noções de classe e interface, bem assim como as duas hierarquias

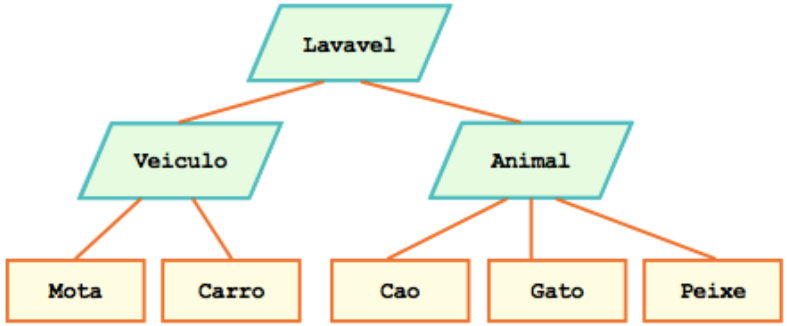
- Uma classe passa a ter duas vistas (ou classificação) possíveis:
  - é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento
  - é subtipo, por se enquadrar numa hierarquia múltipla de definições de comportamento abstracto (puramente sintático)

- Existem situações que apenas são possíveis de satisfazer considerando as duas hierarquias.



- se fosse importante saber os objectos desta hierarquia que poderiam ser lavados, então...

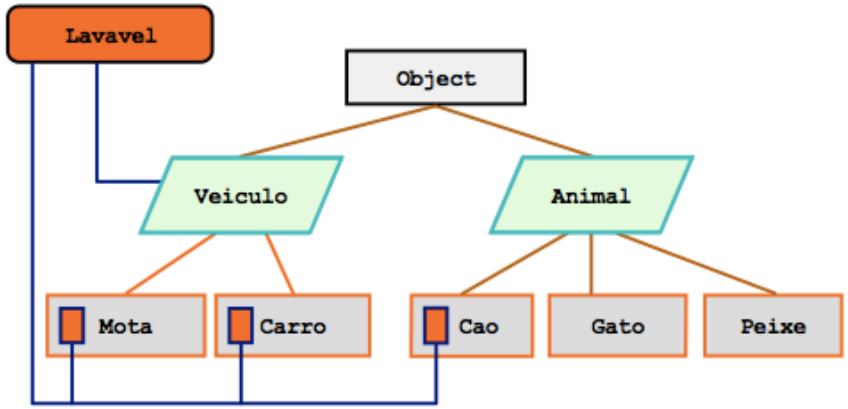
- A hierarquia "correcta" (que dava mais jeito) seria:



- no entanto, esta solução obrigaria objectos não "laváveis", a ter um método `aLavar()`

- Com a utilização de ambas as hierarquias poderemos ter:

```
public interface Lavavel {
   public void aLavar();
}
```



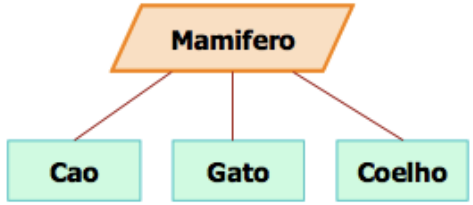
-

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)

Tipos Parametrizados

- Seja a seguinte hierarquia:

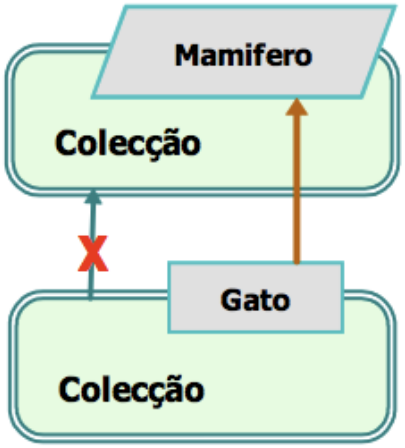


  - e considere-se uma colecção de elementos do tipo Mamífero

- Um arraylist de mamíferos, ArrayList<Mamifero> pode conter instâncias de Cão, Gato, Coelho, etc.

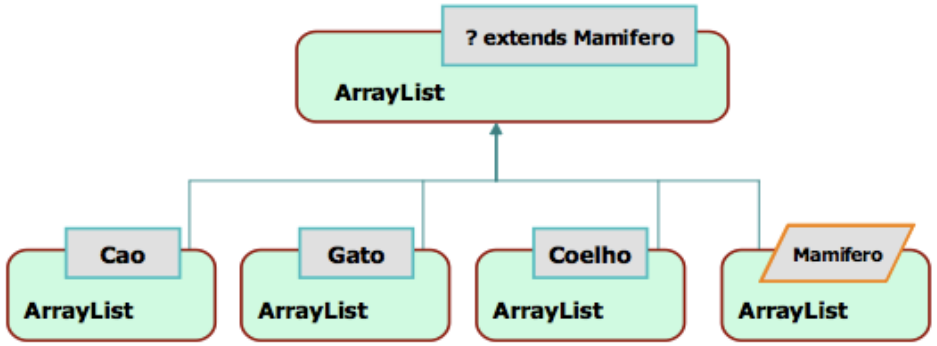
- no entanto esse arraylist não é supertipo dos arraylist de subtipos de mamíferos!!

- A hierarquia de ArrayList<E> não tem a mesma estruturação da hierarquia de E



- o arraylist de todos os arraylists de Mamifero e dos seus subtipos é o arraylist que se declara como:

- ***ArrayList<? extends Mamifero>***




```
Coleccao<? extends Mamifero> =
    Coleccao<Gato> ou
    Coleccao<Cao>  ou
    Coleccao<Coelho>
```

```
public void juntaMamif(Set<? extends Mamifero> cm) {
  ...
}
```

- Desta forma passa a ser possível ter declarações como:

```
ArrayList<Mamifero> mamifs = new ArrayList<Mamifero>();
mamifs.addAll(criaCaes());  // junta ArrayList<Cao>
mamifs.addAll(criaGatos()); // junta ArrayList<Gato>
```

  - o que era impossível no modelo anterior, na medida em que um ArrayList<Cao> não é compatível com ArrayList<Mamifero>

Programação Orientada aos Objectos

-- fecho do semestre --

Resultados de aprendizagem

- Compreender os conceitos fundamentais da PPO(Objectos, Classes, Herança e Polimorfismo);
- Compreender como os conceitos básicos da PPO são implementados em construções JAVA;
- Compreender princípios e técnicas a empregar em programação de larga escala;
- Desenvolver o modelo de classes e interfaces para um dado problema de software (modelação);
- Desenvolver e implementar aplicações Java de média escala, seguras, robustas e extensíveis;

Programa: teórica

- Paradigma da programação por Objectos: Abstracção de Dados, Encapsulamento e Modularidade.
- Objectos: estrutura e comportamento.
- Mensagens. Classes, hierarquia e herança. Classes abstractas.
- Herança versus Composição.
- Compatibilidade de tipos. O princípio da substituição. Dynamic binding. Polimorfismo.

Programa prática

- JAVA: Plataforma J2SE: JDK, JVM e byte-code.
- Construções básicas: tipos primitivos e operadores. Estruturas de controlo. I/O básico. Arrays.
- Nível dos objectos: Classes e instâncias. Construtores. Métodos e variáveis de instância. Modificadores de acesso. Métodos e variáveis de classe.
- Colecções genéricas. Interfaces parametrizadas. Iteradores. Tipos List, Map e Set.
- Hierarquia de classes e herança. Overloading e overriding de métodos. Classes Abstractas. Interfaces e tipos definidos pelo utilizador. Tipo estático e dinâmico. Procura dinâmica de métodos. Polimorfismo e extensibilidade.
- Streams: de caracteres, de bytes e de objectos.
- Excepções.