

Herança vs Composição

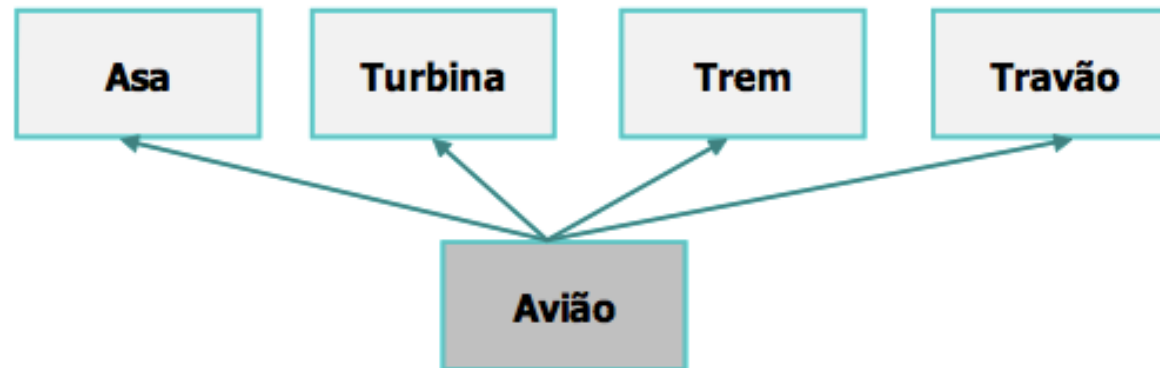
- Herança e composição são duas formas de relacionamento entre classes
- são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas da mesma forma
- existe uma tendência para se confundir herança com composição

- quando uma classe é criada por composição de outras, isso implica que as instâncias das classes agregadas fazem parte da definição do contendor
- 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**

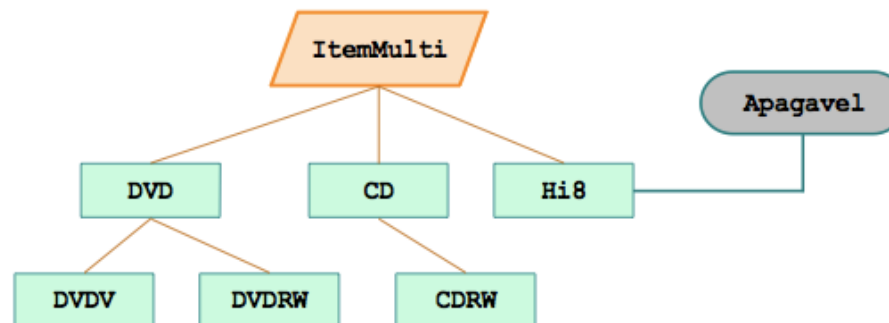
- Consultório de dúvidas

- De acordo com o livro "JAVA6 - programação orientada pelos objectos", encontramos a seguinte frase na pág. 172: "Assim, e de forma geral, a expressão `this.m()`, onde quer que seja encontrada no código de um método de instância de uma dada classe de hierarquia, corresponderá sempre à execução do método `m()` da classe do receptor da mensagem que despoletou o algoritmo de procura."
- No caso de o método `m()` não existir na classe do receptor da mensagem, que despoletou o algoritmo de procura, se é iniciada uma nova procura nas superclasses ou se dá erro. Fazendo alusão ao exemplo da aula teórica, caso a classe `subA` não tivesse o método `getX()` também, que aconteceria? Daria erro ou executava o da classe `Super`?

- De acordo com o livro "JAVA6 - programação orientada pelos objetos", encontramos a seguinte frase na pág. 172: "A mesma regra se aplica para expressões do tipo `super.m()`, em cujo caso a execução do método `m()` é remetida para a superclasse do receptor?
- 2.1 Desta frase deduz-se que o `super.m()` é em tudo igual ao `this.m()` a não ser a pesquisa que começa imediatamente na superclasse da classe receptora da mensagem, assim como a execução do próprio método `m()` agora já não é feita na classe receptora mas sim na respectiva superclasse. Esta afirmação está correta?
- 2.2 A última dúvida é exatamente igual à primeira, mas desta vez com o método `super`. Se o método `m()` não existir na superclasse da classe receptora da mensagem, isto é, a classe onde supostamente o método `m()` seria executado, o que acontece? Dá erro? Inicia-se uma pesquisa na superclasse da superclasse da classe que recebeu a mensagem?

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; }
}
```

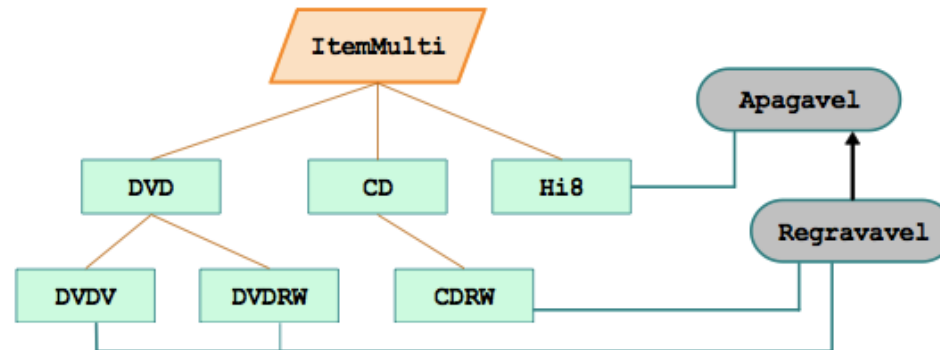
```
public interface Apagavel {
    /**
     * Apagar
     */
    public void apaga();
}
```

- 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.

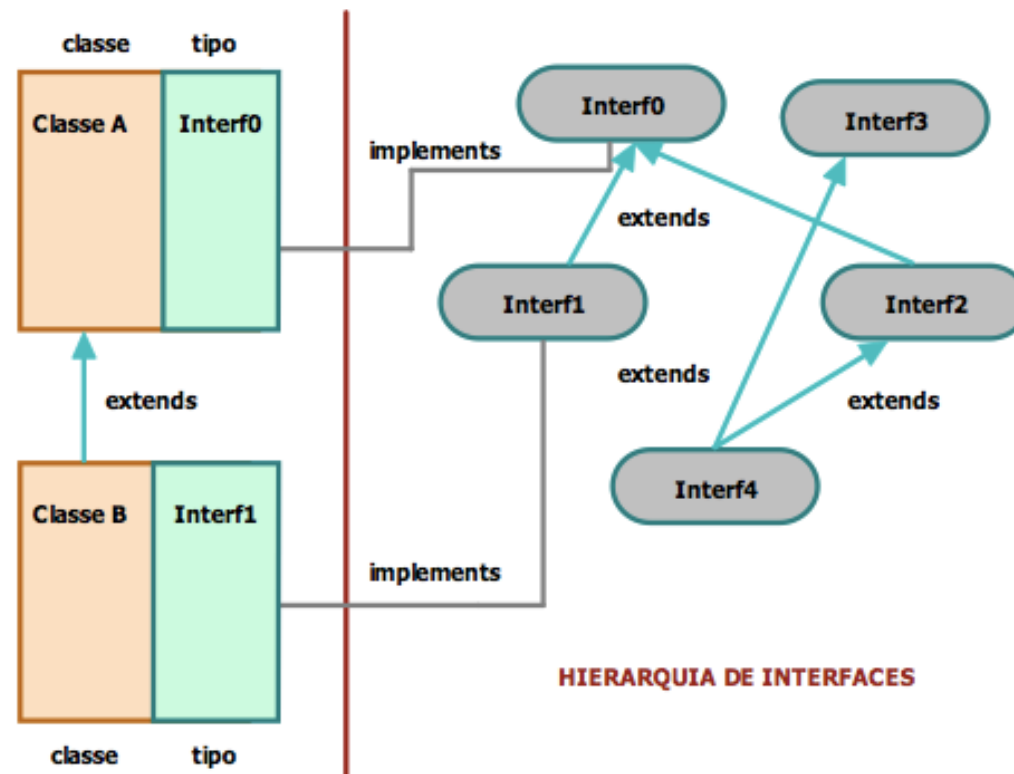
- A verificação de tipo pode ser feita da mesma forma que fazemos para as classes, com instanceof

```
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 (como no caso do JCF)

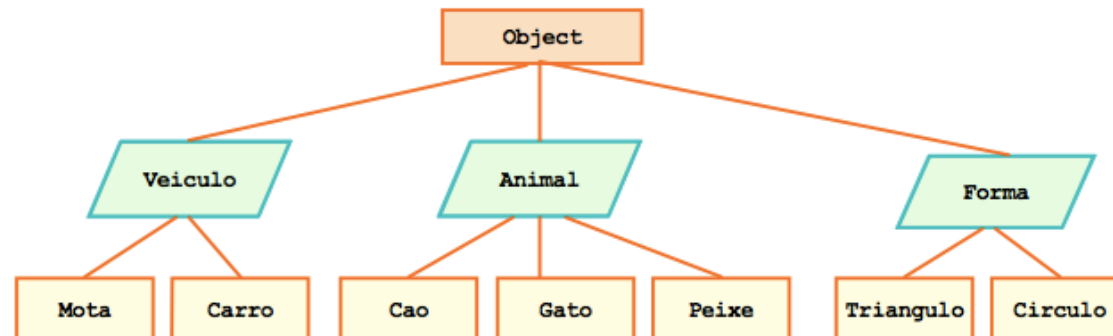
- O modelo geral é assim:



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

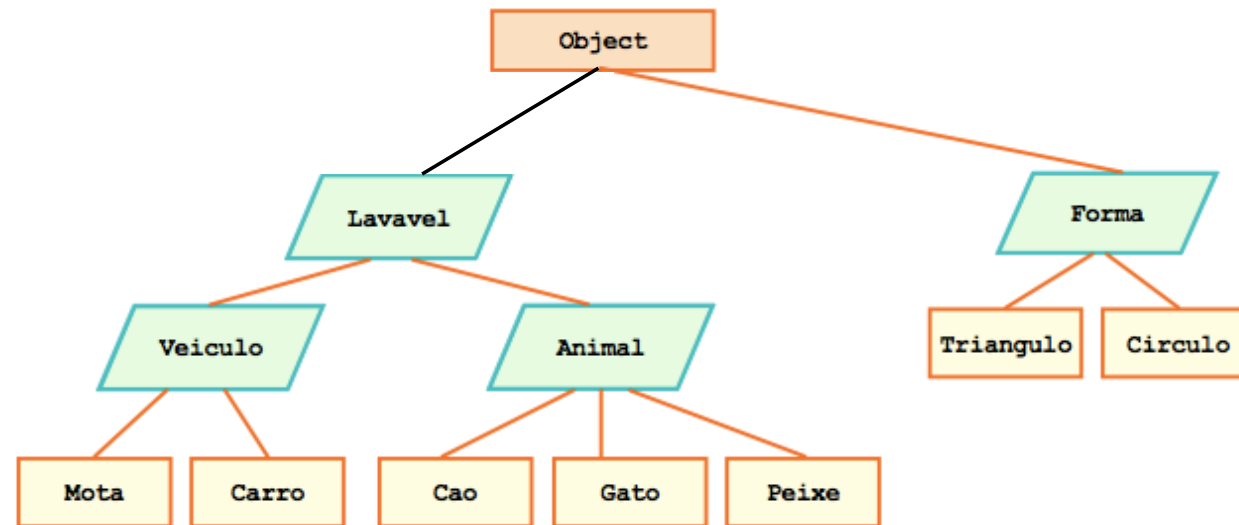
- Uma classe passa a ter duas vistas (ou classificações) possíveis:
- é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento
- é subtipo, por se enquadrar numa hierarquia múltipla de definições de comportamento abstracto (puramente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

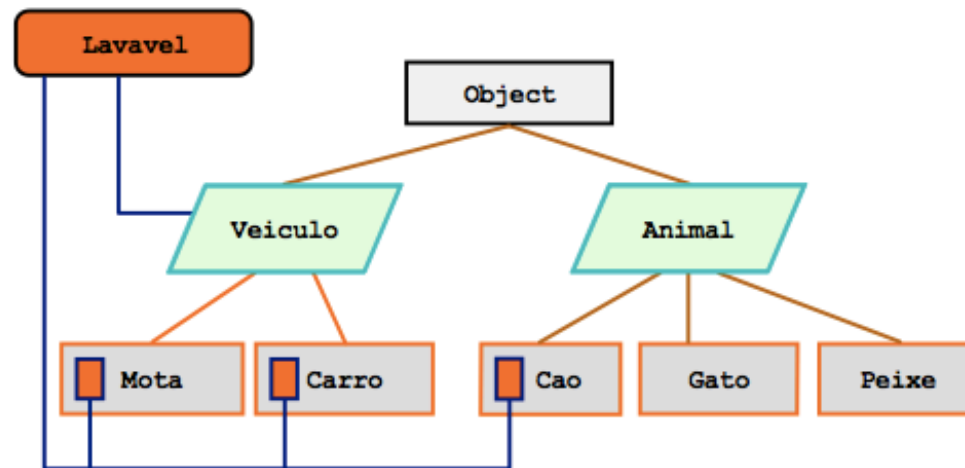
- Podíamos pensar em...



- 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();  
}
```



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)

Tratamento de Erros

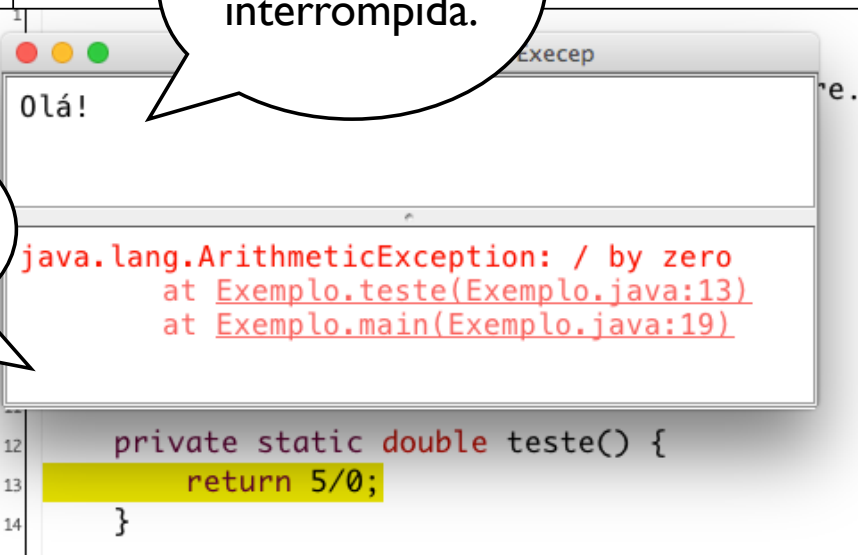
- Java usa a noção de *excepções* para realizar tratamento de erros
- Uma exceção é um *evento* que ocorre durante a execução do programa e que interrompe o fluxo normal de processamento

Exceções

```
10 public class Exemplo {  
11  
12     private static double teste() {  
13         return 5/0;  
14     }  
15  
16  
17     public static void main(String[] args) {  
18         System.out.println("Olá!");  
19         System.out.println(teste());  
20         System.out.println("Até logo!");  
21     }  
22 }  
23 }
```

O erro é propagado para trás pela stack de invocações de métodos.

A execução é interrompida.



```
java.lang.ArithmeticException: / by zero  
at Exemplo.teste(Exemplo.java:13)  
at Exemplo.main(Exemplo.java:19)  
  
private static double teste() {  
    return 5/0;  
}
```


try catch

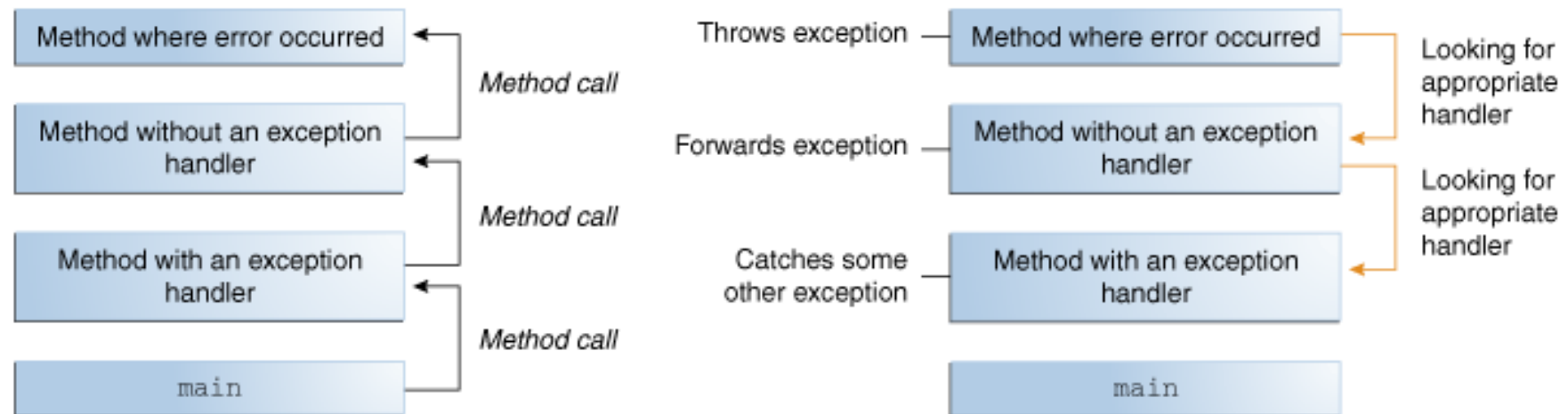
```
10 public class Exemplo {  
11  
12     private static double teste() {  
13         return 5/0;  
14     }  
15  
16  
17     public static void main(String[] args) {  
18         System.out.println("Olá!");  
19         try {  
20             System.out.println(teste());  
21         }  
22         catch (ArithmeticException e) {  
23             System.out.println("Ops! "+e.getMessage());  
24         }  
25         System.out.println("Até logo!");  
26     }  
27 }
```

A execução
retoma no **catch**.

BlueJ: Terminal Window - Execep

```
Olá!  
Ops! / by zero  
Até logo!
```

Excepções



Criar Exceções

```
public class AlunoException extends Exception {  
    public AlunoException(String msg) {  
        super(msg);  
    }  
}
```

```
/**  
 * Obter o aluno da turma com número num.  
 *  
 * @param num o número do aluno pretendido  
 * @return uma cópia do aluno na posição  
 * @throws AlunoException  
 */  
public Aluno getAluno(int num) throws AlunoException {  
    Aluno a = alunos.get(num);  
    if (a==null)  
        throw new AlunoException("Aluno "+num+" não existe");  
    return a.clone();  
}
```

Lança uma exceção.

Obrigatório declarar que lança exceção.

```
public static void main(String[] args) {  
    Opcoes op;  
    Aluno a;  
    int num;  
    do {  
        op = lerOpcao();  
        switch (op) {  
            CONSULTAR:  
            num = leNumero();  
            try {  
                a = turma.getAluno(num);  
                out.println(a.toString());  
            }  
            catch (AlunoException e) {  
                out.println("Ops "+e.getMessage());  
            }  
            break;  
            INSERIR:  
            ...  
        }  
    } while (op != Opcoes.SAIR);  
}
```

Vai tentar um getAluno...

Apanha e trata a exceção.

Tipos de Exceções

- Exceções de *runtime*
 - Condições excepcionais interna à aplicação - ou seja, bugs!!
 - **RuntimeException** e suas subclasses
 - Exemplo; **NullPointerException**
- Erros
 - Condições excepcionais externas à aplicação
 - **Error** e suas subclasses
 - Exemplo: **IOError**
- Checked Exceptions
 - Condições excepcionais que aplicações bem escritas deverão tratar
 - Obrigadas ao requisito *Catch or Specify*
 - Exemplo: **FileNotFoundException**

Vantagens do uso de Exceções

- Separam código de tratamento de erros o código *regular*
- Propagação dos erros pela stack the invocações de métodos
- Junção e diferenciação de tipos de erros

Exemplo

Leitura/Escrita em ficheiros

EmpresaPOO

```
public void gravaObj(String fich) throws IOException {
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fich));
    oos.writeObject(this);
    oos.flush();
    oos.close();
}

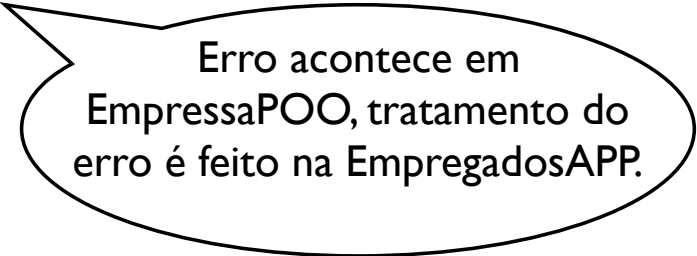
public static EmpresaPOO leObj(String fich) throws IOException, ClassNotFoundException {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fich));

    EmpresaPOO te= (EmpresaPOO) ois.readObject();
    ois.close();
    return te;
}

public void log(String f, boolean ap) throws IOException {
    FileWriter fw = new FileWriter(f, ap);
    fw.write("\n----- LOG - LOG - LOG - LOG - LOG ----- \n");
    fw.write(this.toString());
    fw.write("\n----- LOG - LOG - LOG - LOG - LOG ----- \n");
    fw.flush();
    fw.close();
}
```

EmpregadosApp

```
private static void carregarDados() {  
    try {  
        tab = EmpresaPOO.leObj("estado.tabemp");  
    }  
    catch (IOException e) {  
        tab = new EmpresaPOO();  
        System.out.println("Não consegui ler os dados!\nErro de leitura.");  
    }  
    catch (ClassNotFoundException e) {  
        tab = new EmpresaPOO();  
        System.out.println("Não consegui ler os dados!\nFicheiro com formato desconhecido.");  
    }  
    catch (ClassCastException e) {  
        tab = new EmpresaPOO();  
        System.out.println("Não consegui ler os dados!\nErro de formato.");  
    }  
}
```



Erro acontece em
EmpresaPOO, tratamento do
erro é feito na EmpregadosAPP.

```

public static void main(String[] args) {
    carregarMenus();
    carregarDados();
    do {
        menumain.executa();
        switch (menu.getOpcao()) {
            case 1: inserirEmp();
                    break;
            case 2: consultarEmp();
                    break;
            case 3: totalSalarios();
                    break;
            case 4: totalGestores();
                    break;
            case 5: totalPorTipo();
                    break;
            case 6: totalKms();
                    break;
        }
    } while (menu.getOpcao() != 0);
    try {
        tab.gravaObj("estado.tabemp");
        tab.log("log.txt", true);
    }
    catch (IOException e) {
        System.out.println("Não consegui gravar os dados!");
    }
    System.out.println("Até breve!...");
}

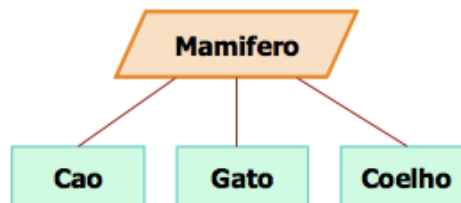
```

Carregar dados no início
(erros são tratados dentro de
carregarDados).

Gravar dados
(e log) no fim (erros são
tratados aqui).

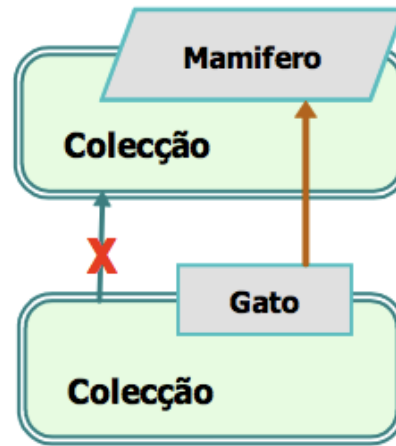
Tipos Parametrizados

- Seja a seguinte hierarquia:

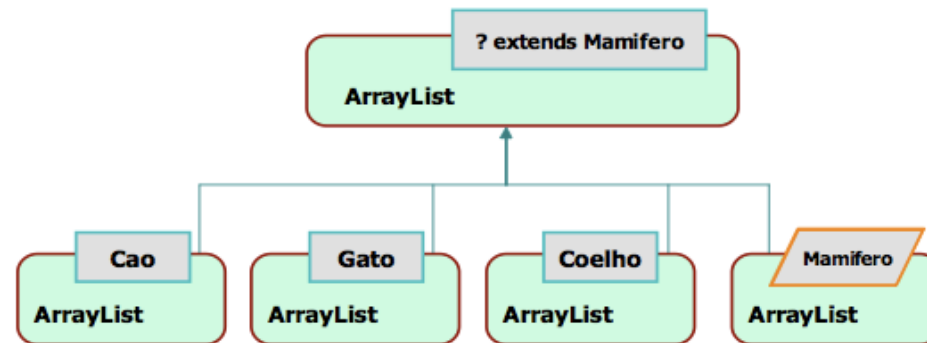


- e considere-se uma colecção de elementos do tipo Mamífero

- 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 tipo dos ArrayLists de Mamifero e dos ArrayLists seus subtipos declara-se como:
`ArrayList<? extends Mamifero>`
- O tipo dos ArrayLists de super-classes de Mamifero declara-se como:
`ArrayList<? super 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>`