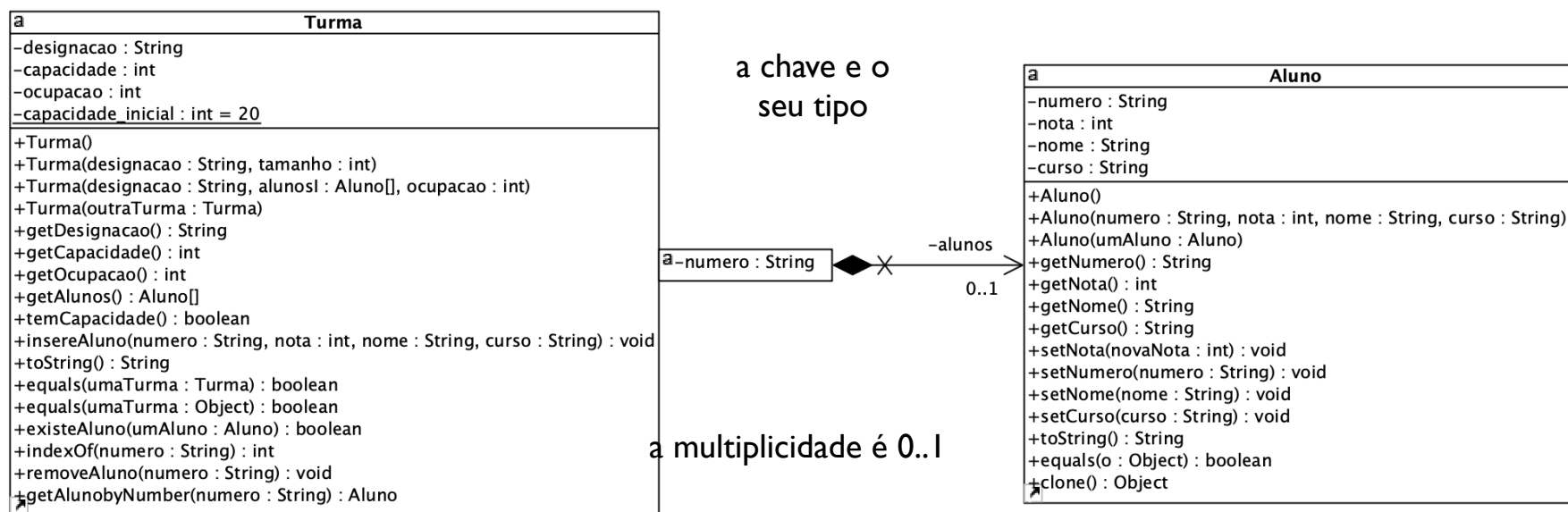


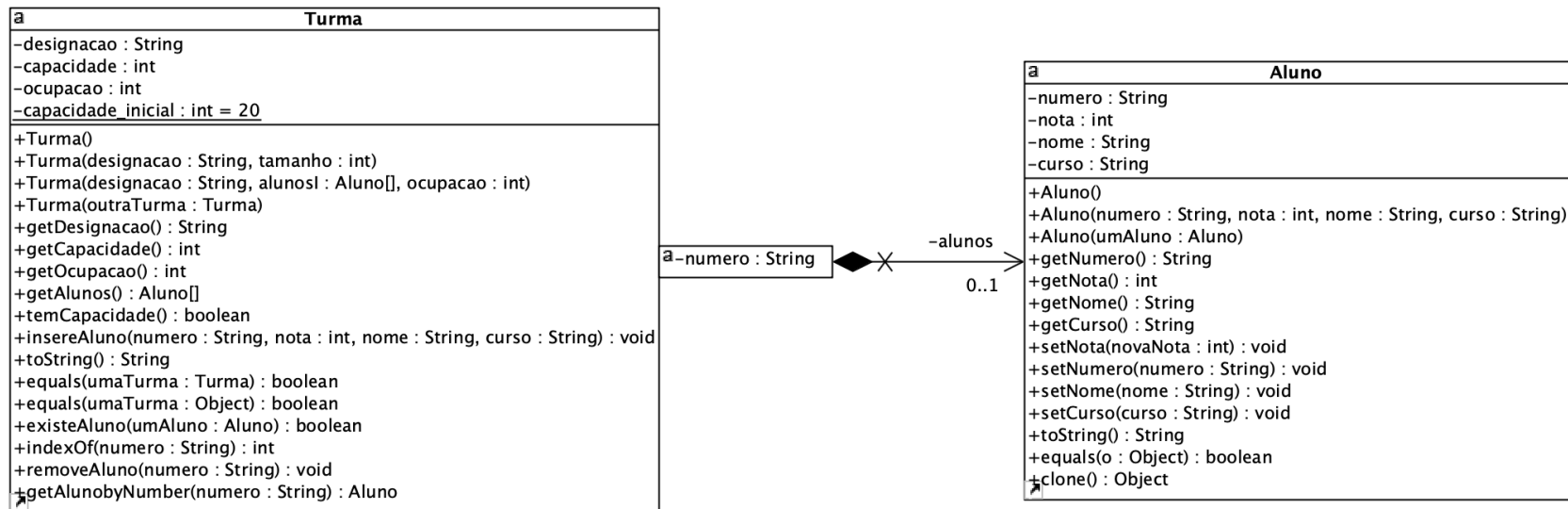
# Notação do Diagrama de Classe UML

- Em relação ao diagrama de classe UML que temos vindo a utilizar é necessário acrescentar mais informação:
  - como descrever apropriadamente mapeamentos
  - como representar implementação de interfaces
  - como descrever o que é abstracto

- Na descrição do Map vamos indicar a chave e o seu tipo e a classe dos objectos que fazem parte dos valores.



- se a chave existir temos acesso a uma instância de Aluno, caso contrário a zero!



## ● dá origem a:

```

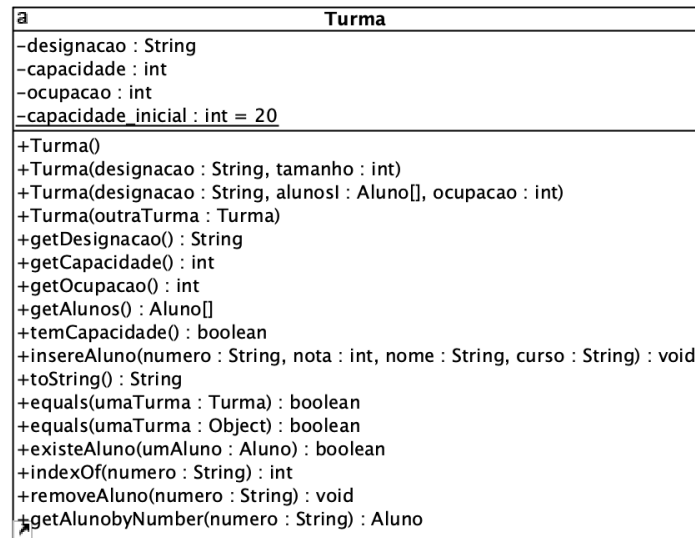
public class Turma {
    private String designacao;
    private int capacidade;
    private int ocupacao;
    private static final int capacidade_inicial = 20;

    private Map<String, Aluno> alunos;

    ...
}
  
```

- Em UML as definições que são abstractas são anotadas a *itálico* ou em alguns editores como utilizando a notação <<abstract>> (já não faz parte da norma...)

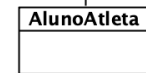
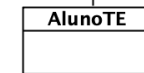
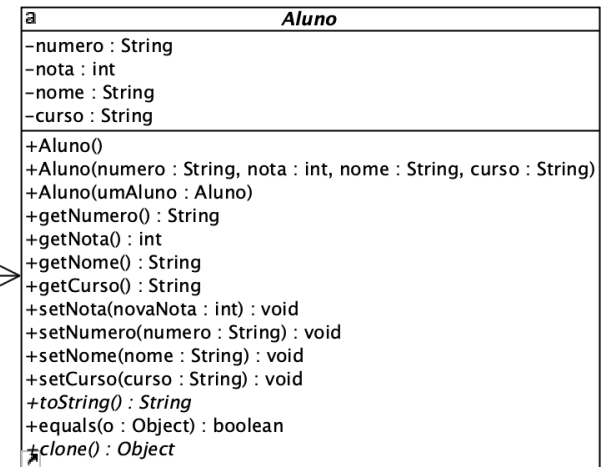
Aluno é  
abstract



a-numero : String

-alunos  
0..1

toString e  
clone são  
abstract



AlunoTe e  
AlunoAtleta implementam  
a interface ComRegime

# Tratamento de Erros

- Nesta altura já se apresentou uma forma de efectuar testes unitários, isto é fazer programas que fazem asserções sobre o comportamento exibido pelos programas em determinadas situações.
- Esses testes podem ser executados sempre que se altera o código como medida preventiva de detecção de erros antes de passarmos o componente (classe, módulo, etc.) para terceiros.

- Criar o teste, construir os objectos...

```
public TestFeed() {  
  
    FBPost post0 = new FBPost(0, "User 1", LocalDateTime.of(2018,3,10,10,30,0), "Teste 1", 0, new ArrayList<  
    FBPost post1 = new FBPost(1, "User 1", LocalDateTime.of(2018,3,12,15,20,0), "Teste 2", 0, new ArrayList<  
    FBPost post2 = new FBPost(2, "User 2", LocalDateTime.now(), "Teste 3", 0, new ArrayList<>());  
    FBPost post3 = new FBPost(3, "User 3", LocalDateTime.now(), "Teste 4", 0, new ArrayList<>());  
    FBPost post4 = new FBPost(4, "User 4", LocalDateTime.now(), "Teste 5", 0, new ArrayList<>());  
  
    List<FBPost> tp = new ArrayList<>();  
    tp.add(post0);  
    tp.add(post1);  
    tp.add(post2);  
    tp.add(post3);  
    tp.add(post4);  
    //tp.add(post5);  
    feed.setPosts(tp);  
}
```

```
@Test
public void testNrPosts() {
    int np = feed.nrPosts("User 1");
    assertEquals(np,2);
    //assertTrue(np == 2);
}
```

```
@Test
public void testPostsOf() {
    List<FBPost> posts = feed.postsOf("User 2");
    assertNotNull(posts);
    assertEquals(posts.size(),1);
    FBPost p = feed.postsOf("User 2").get(0);
    assertNotNull(p);
    assertEquals("User 2",p.getUsername());
}
```



```
@Test
public void testGetPost() {
    FBPost p = feed.getPost(3);
    assertEquals(p.getUsername(), "User 3");
}
```

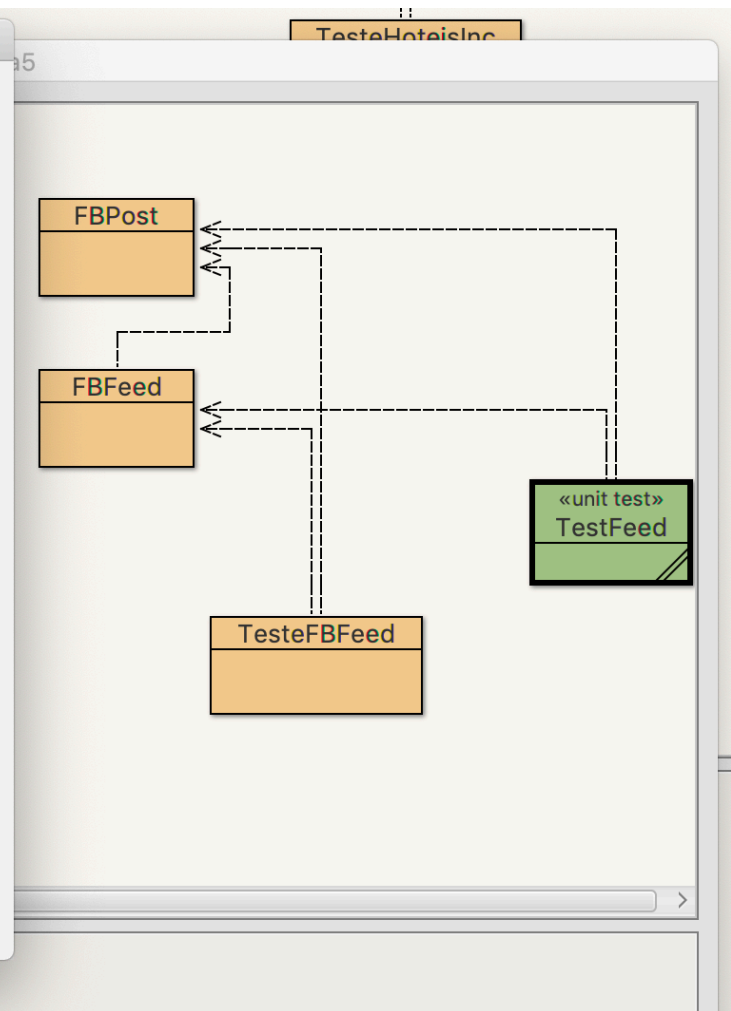
```
@Test
public void testComment() {
    FBPost p = feed.getPost(3);
    feed.comment(p, "Primeiro comentario");
    assertTrue(p.getComentarios().size() == 1);
    assertEquals(p.getComentarios().get(0), "Primeiro comentario");
}
```

BlueJ: Test Results

- ✓ TestFeed.testLike
- ✓ TestFeed.testTop5
- ✓ TestFeed.testPostsOf
- ✓ TestFeed.testeClasse
- ✓ TestFeed.testGetPost
- ✓ TestFeed.testPostsOfDate
- ✓ TestFeed.testComment
- ✓ TestFeed.testNrPosts

Runs: 8/8    ✗Errors:0    ✗Failures:0    Total Time: 43ms

Show Source    Close



- Existem outro tipo de testes que se designam por testes de integração, onde é suposto fazermos uma avaliação qualitativa da orquestração dos vários objectos no meu programa.
- poderá acontecer que uma classe não apresente problemas quando testada de forma unitária, mas ao ser integrada numa outra classe apresente problemas.

- Podemos olhar para cada um dos métodos que são oferecidos pelas classes (e pelas interfaces) como sendo contratos. E neles poder especificar:
  - as condições em que podem ser invocados
  - o que realizam (o algoritmo)
  - o que acontece depois de serem executados, isto é, o que aconteceu ao estado

- Podemos determinar asserções sobre:
  - as pré-condições, o que tem de ser garantido para que o método possa ser executado
  - as pós-condições, a validação das alterações ao estado em caso de sucesso da execução correcta do método

- Contudo, nem sempre é possível executar um método. Por exemplo:
  - criar um círculo de raio negativo
  - levantar dinheiro de uma conta sem saldo suficiente
  - efectuar uma viagem com distância superior à autonomia do veículo

- Nessas circunstâncias, o método deve enviar um sinal de erro.
- numa lógica diferente dos erros do C
- que obriguem o erro a ser verificado
- que se possam efectuar operações de gestão do erro (acções de recuperação).

- (das fichas...) Temos escrito código e comentado situações em que o comportamento pode não ser o esperado.

classe que  
implementa Map.Entry

e  
se V não existe?

```
import static java.util.AbstractMap.SimpleEntry;
import static java.util.Map.Entry;

...
// Dá erro se vértice não existe
Set<Entry<String, String>> fanOut (String v) {
    Set<Entry<String, String>> res = new HashSet<>(); // SimpleEntry não é Comparable!

    for (String vout: this.adj.get(v)) {
        res.add(new SimpleEntry<>(v, vout));
    }
    return res;
}

Set<Entry<String, String>> fanIn(String v) {
    Set<Map.Entry<String, String>> res = new HashSet<>(); // SimpleEntry não é Comparable!

    for (Entry<String, Set<String>> e: this.adj.entrySet()) {
        if (e.getValue().contains(v)) {
            res.add(new SimpleEntry<>(e.getKey(), v));
        }
    }
    return res;
}
```



# Tratamento de Erros

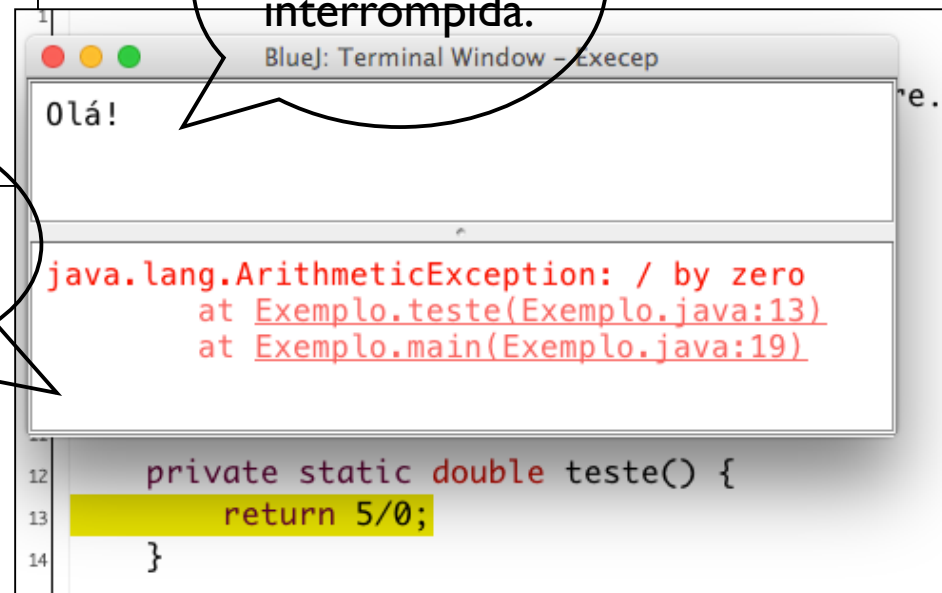
- Java usa a noção de *exceções* para realizar tratamento de erros
- Uma exceção é um *evento* que ocorre durante a execução do programa e que interrompe o fluxo normal de processamento
- garante-se assim que o surgimento de um erro obriga o programador a criar código para o tratar. Em vez de apenas o ignorar...

# 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.



The screenshot shows a BlueJ terminal window titled "BlueJ: Terminal Window - Execep". The output shows "Olá!" followed by a red error message: "java.lang.ArithmeticException: / by zero at Exemplo.teste(Exemplo.java:13) at Exemplo.main(Exemplo.java:19)". Below the error message, the code for the `teste()` method is shown, with the line `return 5/0;` highlighted in yellow.

```
12 private static double teste() {  
13     return 5/0;  
14 }
```

# try e 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 }
```

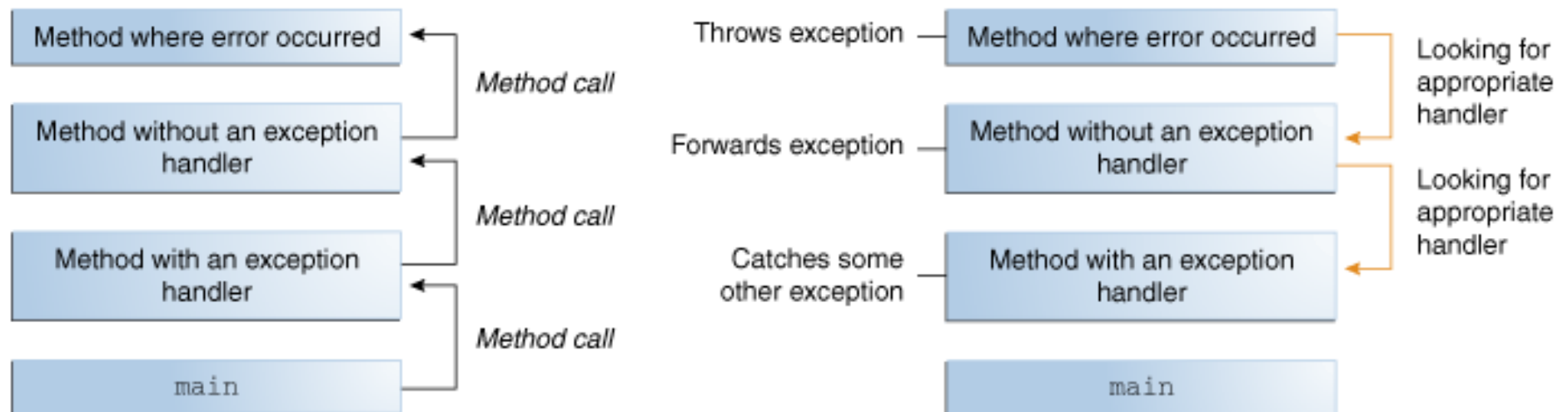
A execução  
retoma no **catch**.

BlueJ: Terminal Window - Execep

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

# Exceções

- Modelo de funcionamento:



# 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 referida  
 * @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, erros nossos!!
  - **RuntimeException** e suas subclasses
  - Exemplo: **NullPointerException**
- Erros
  - Condições excepcionais externas à aplicação
  - **Error** e suas subclasses
  - Exemplo: **IOException**
- Checked Exceptions
  - Condições excepcionais que aplicações bem escritas deverão tratar
  - Obrigadas ao requisito *Catch or Specify*
  - Exemplo: **FileNotFoundException**

# Modelo de utilização das exceções

- Os métodos onde são detectadas as exceções devem sinalizar isso (`throws ...Exception`)
- recomenda-se que para cada tipo de exceção se crie uma classe de Excepção
- métodos que invocam métodos que libertam exceções devem decidir se as tratam ou fazem passagem das mesmas (`throws ...Exception`)

- Se não for feito antes, o tratamento das excepções chega ao método `main()`
- aí pode ser feita toda a gestão da comunicação com o utilizador (`out.println` ou outras)
- métodos de outras classes, que não a classe de teste, não devem enviar informação de erro para o écran.



# Vantagens do uso de Exceções

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

# Exemplo

## Leitura/Escrita em ficheiros

- Gravar em modo texto:

```
/**
 * Método que guarda o estado de uma instância num ficheiro de texto.
 *
 * @param nome do ficheiro
 */
public void escreveEmFicheiroTxt(String nomeFicheiro) throws IOException {
    PrintWriter fich = new PrintWriter(nomeFicheiro);
    fich.println("----- HotéisInc -----");
    fich.println(this.toString()); // ou fich.println(this);
    fich.flush();
    fich.close();
}
```

- Gravação modo binário:
  - obrigatório decidir que classes são persistidas através da implementação da interface `Serializable`
- utilização de `java.io.ObjectOutputStream`

necessita de implements  
`Serializable`

```
/**
 * Método que guarda em ficheiro de objectos o objecto que recebe a mensagem.
 */

public void guardaEstado(String nomeFicheiro) throws FileNotFoundException, IOException {
    FileOutputStream fos = new FileOutputStream(nomeFicheiro);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(this); //guarda-se todo o objecto de uma só vez
    oos.flush();
    oos.close();
}
```

- Leitura em modo binário
- utilização de  
`java.io.ObjectInputStream`

```
/**
 * Método que recupera uma instância de HoteisInc de um ficheiro de objectos.
 * Este método tem de ser um método de classe que devolva uma instância já
 * construída de HoteisInc.
 *
 * @param nome do ficheiro onde está guardado um objecto do tipo HoteisInc
 * @return objecto HoteisInc inicializado
 */
public static HoteisInc carregaEstado(String nomeFicheiro) throws FileNotFoundException,
                                     IOException,
                                     ClassNotFoundException {
    FileInputStream fis = new FileInputStream(nomeFicheiro);
    ObjectInputStream ois = new ObjectInputStream(fis);
    HoteisInc h = (HoteisInc) ois.readObject();
    ois.close();
    return h;
}
```

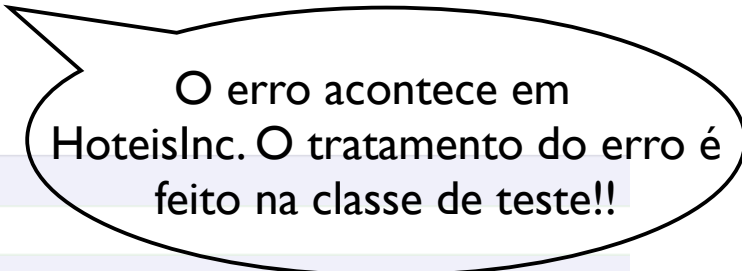
# Utilização na classe de teste

```
//Gravar em ficheiro de texto
```

```
try {  
    osHoteis.escreveEmFicheiroTxt("estadoHoteisTXT.txt");  
}  
catch (IOException e) {System.out.println("Erro a aceder a ficheiro!");}
```

```
//Gravar em ficheiro de objectos
```

```
try {  
    osHoteis.guardaEstado("estadoHoteis.obj");  
}  
catch (FileNotFoundException e) {  
    System.out.println("Ficheiro não encontrado!");  
}  
catch (IOException e) {  
    System.out.println("Erro a aceder a ficheiro!");}
```



O erro acontece em HoteisInc. O tratamento do erro é feito na classe de teste!!

```

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

```

Carregar dados no início  
(erros são tratados dentro do  
método de carregamento).

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

# A abordagem do nio

- As classes apresentadas atrás permitem, de forma simples, ter uma estratégia de utilização das inúmeras streams para persistência de informação
- para gravar em texto: `PrintWriter`
- para gravar em modo binário: `ObjectOutputStream`
- para ler em modo binário: `ObjectInputStream`

# A classe Files

- Na classe Files (`nio.File.Files`) encontram-se muitos métodos disponíveis para operações sobre ficheiros (e gestão do sistema de ficheiros)
- Possui métodos de âmbito mais geral sobre ficheiros, possibilitando operações de mais alto nível, quer na leitura quer no acesso à informação.



# A classe Files

- Exemplo disso é a utilização de `lines(Path p)` ou de `readAllLines(Path p)` para leitura em bulk de dados de um ficheiro de texto.
- a estratégia é depois utilizar-se um mecanismo de parsing das `String` obtidas para encontrar a informação pretendida
- Um exemplo é o fornecimento de informação para o carregamento do ficheiro de dados iniciais do projecto.

- O método `readAllLines` devolve uma `List<String>`
- Em `Path` deve ser passado o caminho para o ficheiro

```
public List<String> lerFicheiro(String nomeFich) {  
    List<String> lines = new ArrayList<>();  
    try { lines = Files.readAllLines(Paths.get(nomeFich), StandardCharsets.UTF_8); }  
    catch(IOException exc) { System.out.println(exc.getMessage()); }  
    return lines;  
}
```

- Obtêm-se uma `List<String>` com o resultado das várias linhas do ficheiro (que tem de ter linefeed) e depois interpreta-se cada linha sabendo qual o separador (":")

```
public static void parse() throws LinhaIncorretaException {
    List<String> linhas = lerFicheiro("C:\\Path\\To\\file.csv");
    Map<String, Fornecedor> fornecedores = new HashMap<>();
    Map<Integer, CasaInteligente> casas = new HashMap<>();
    String[] linhaPartida;
    CasaInteligente c = null; SmartDevice sd = null;
    String divisao = null;
    for (String linha : linhas) {
        linhaPartida = linha.split(":", 2);
        switch(linhaPartida[0]){
            case "Fornecedor":
                Fornecedor f = Fornecedor.parse(linhaPartida[1]);
                fornecedores.put(f.getFornecedor(), f);
                break;
            case "Casa":
                c = CasaInteligente.parse(linhaPartida[1]);
                casas.put(c.getNif(), c);
                divisao = null;
                break;
            case "Divisao":
                if (c == null) throw new LinhaIncorretaException();
                divisao = linhaPartida[1];
                c.addRoom(divisao);
                break;
            case "SmartBulb":
                if (divisao == null) throw new LinhaIncorretaException();
                sd = SmartBulb.parse(linhaPartida[1]);
                c.addDevice(sd);
                c.addToRoom(divisao, sd.getId());
                break;
            case "SmartCamera":
                if (divisao == null) throw new LinhaIncorretaException();
                sd = SmartCamera.parse(linhaPartida[1]);
                c.addDevice(sd);
                c.addToRoom(divisao, sd.getId());
                break;
        }
    }
}
```

# O package java.util.function

- Este package possui várias interfaces funcionais que foram adicionadas e cujo objectivo é poderem ser utilizadas para parametrizar as classes através da injeção de comportamento.
- São interfaces funcionais cuja definição contém apenas um método.
- que é abstracto e será instanciado pelo programador.

(\*) seguindo a estrutura apresentada em Functional Interfaces in Java, Ralph Lecessi, 2019

- Considerem-se os quatros tipos básicos de entidades deste package:

Model	Has Arguments	Returns a Value	Description
<b>Predicate</b>	yes	boolean	Tests argument and returns true or false.
<b>Function</b>	yes	yes	Maps one type to another.
<b>Consumer</b>	yes	no	Consumes input (returns nothing).
<b>Supplier</b>	no	yes	Generates output (using no input).

- Estas definições são utilizadas criando-se uma função de um destes tipos de dados e definindo-a utilizando uma expressão lambda.

# Predicate<T>

- A interface `Predicate` faz a avaliação de uma condição associada a um valor de entrada que é de um tipo genérico.
- O método devolve `true` se a condição for verdade, falso caso contrário

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

- O programador associa depois um tipo de dados quando define o predicado:

```
Predicate<Integer> p1;
```

- e faz a associação da expressão que representa a condição. Exemplo:

```
p1 = x -> x > 7;
```

- testando a veracidade com a invocação de test

```
if (p1.test(9))  
    System.out.println("Predicate x > 7 é verdade para x == 9.");
```

- Claro que uma mais valia desta abordagem é poder passar para um método um predicado
- invocar o teste do predicado dentro do método

```
public static void result(Predicate<Integer> p, Integer arg) {  
    if (p.test(arg))  
        System.out.println("O predicado é true para " + arg);  
    else  
        System.out.println("O predicado é falso para " + arg);  
}
```

```
public static void main(String[] args) {  
    Predicate<Integer> p1 = x -> x == 5;  
  
    result(p1, 5);  
    result(y -> y%2 == 0, 5);  
}
```



- Num exemplo de uma das aulas práticas em que temos uma Turma de Alunos, podemos definir um método mais geral que permita identificar os alunos que satisfazem determinado predicado:

```
/**
 * Passar um predicado para um método, possibilitando assim a parametrização
 * de comportamento através de um parâmetro.
 * Este método devolve a lista dos alunos que satisfazem o predicado p
 */

public List<Aluno> alunosqueRespeitamP(Predicate<Aluno> p) {
    return this.alunos.values().
        stream().
        filter(a -> p.test(a)).
        map(Aluno::clone).
        collect(Collectors.toList());
}
```

generaliza-se o mecanismo de filtragem

- Definindo agora os predicados podemos obter diversos filtros de informação:
- não tendo que repetir código
- parametrizando o comportamento de filtragem pretendido

```
Predicate<Aluno> p = a -> a instanceof AlunoTE;  
  
List<Aluno> alunosTE = t.alunosqueRespeitamP(p);  
for (Aluno a: alunosTE)  
    System.out.println(a.toString());
```

# Function<T,R>

- Esta interface funcional aceita dois tipos de dados, sendo que recebe um parâmetro  $T$  e devolve um resultado do tipo  $R$ .

```
@FunctionalInterface
public interface Function<T, R>
{
    R apply(T t);
    ...
}
```

- o método `apply` transforma o objecto do tipo  $T$  numa resposta do tipo  $R$ .

- Na definição da Function é necessário associar os tipos de dados e depois definir o seu comportamento através de uma lambda expression

```
Function<String, Integer> f;  
f = x -> Integer.parseInt(x);
```

- a utilização faz-se pela invocação na Function do método `apply`

```
Integer i = f.apply("100");  
System.out.println(i);
```

**OUTPUT:**

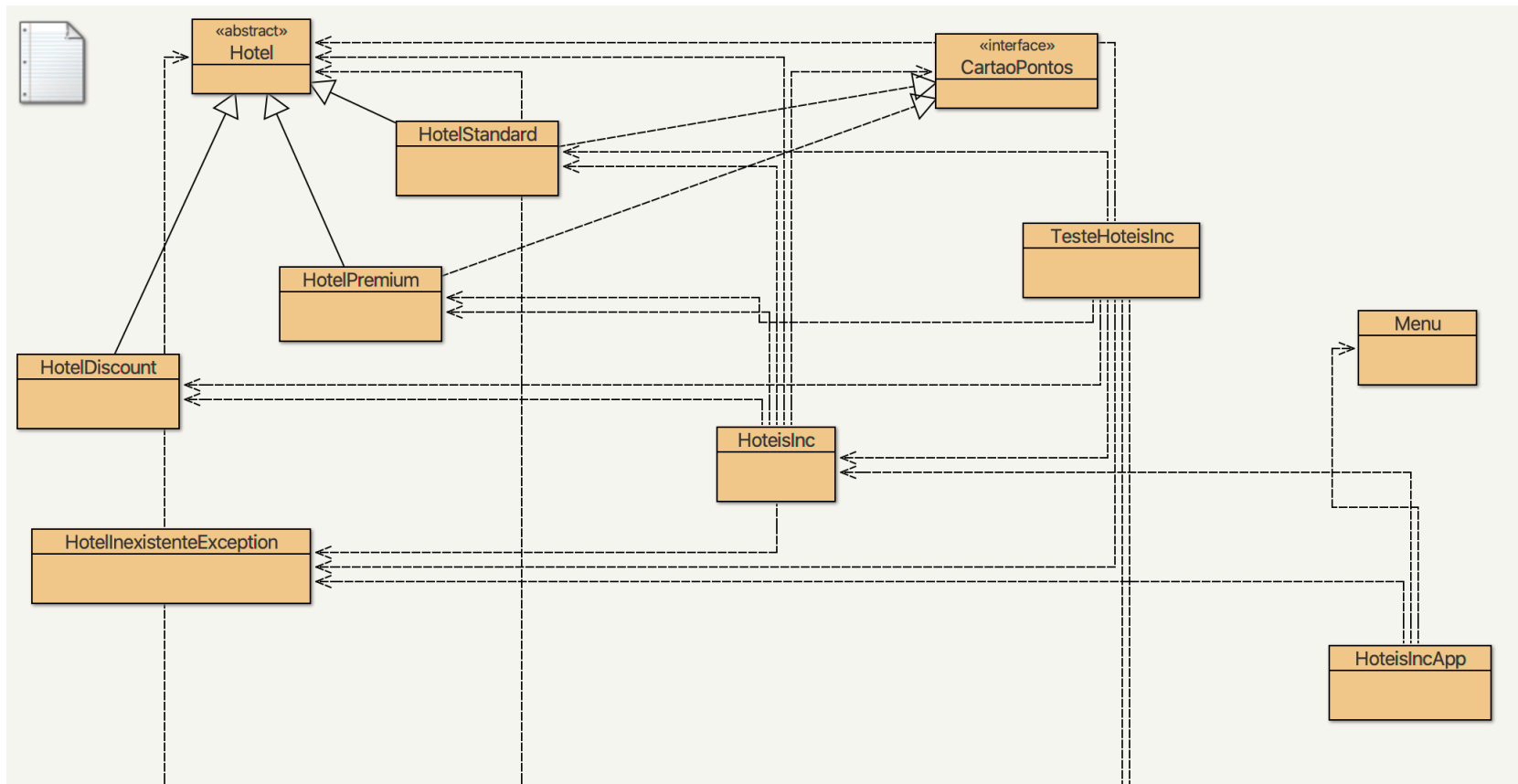
100

- Como vimos para os predicados é possível passar estas Function como parâmetro

```
public class Transformer {  
  
    private static <T,R> R transform(T t, Function<T,R> f) {  
        return f.apply(t);  
    }  
  
    public static void main(String[] args) {  
  
        Function<String,Integer> fsi = x -> Integer.parseInt(x);  
        Function<Integer,String> fis = x -> Integer.toString(x);  
  
        Integer i = transform("100", fsi);  
        String s = transform(200, fis);  
        System.out.println(i);  
        System.out.println(s);  
    }  
}
```

- Podemos utilizar a declaração de Function
  - para tornar mais genéricos todos os métodos que fazem travessias a colecções e aplicam uma função
  - evita-se a repetição de código
  - tal é possível derivado do facto de que é permitido passar uma expressão lambda como parâmetro

- Seja o seguinte projecto:



- Consideremos que queremos fazer diferentes métodos:
- obter o preço de todos os hotéis da cadeia de hotéis
- listar o nome de todos os hotéis
- listar o número de estrelas de todos os hotéis
- Todos estes métodos vão ter um código muito semelhante.



- Pode ser definido um método que aplique a função a todos os objectos do tipo Hotel

```
/**
 * Método que recebe uma Function<T,R> e aplica a todos os
 * hotéis existentes.
 */
public <R> List<R> devolveInfoHoteis(Function<Hotel,R> f) {
    List<R> res = new ArrayList<>();
    for(Hotel h: this.hoteis.values())
        res.add(f.apply(h));

    return res;
}
```

- E, de cada vez, que seja necessário aplicar um novo tipo de selecção de informação criamos uma `Function`
- Exemplo:

```
Function<Hotel,Double> fpreco = h -> h.precoNoite();  
Function<Hotel,String> fnome = h -> h.getNome();
```

```
List<Double> precos = osHoteis.devolveInfoHoteis(fpreco);  
for(Double d: precos)  
    System.out.println(d.toString());
```

```
List<String> nomes = osHoteis.devolveInfoHoteis(fnome);  
for(String d: nomes)  
    System.out.println(d.toString());
```

- Existe também a possibilidade de definir funções binárias, do tipo `Function<T,U,R>`

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
    R apply(T t, U u);
}
```

- apesar de terem tipos `T` e `U`, poderão representar o mesmo tipo de dados.

# Consumer<T>

- Esta interface é utilizada para processamento de informação
- não devolve resultado, é `void`, e aplica o método `accept` a todos os objectos

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);
    ...
}
```

- Podemos também generalizar muitos métodos que fazem travessias e modificam os visitados

```
/**  
 * Método que recebe uma Consumer<T> e aplica a todos os  
 * hotéis existentes.  
 */  
  
public void aplicaTratamento(Consumer<Hotel> c) {  
    this.hoteis.values().forEach(h -> c.accept(h));  
}
```

```
Consumer<Hotel> downgradeEstrelas = h -> h.setEstrelas(h.getEstrelas()-1);  
osHoteis.aplicaTratamento(downgradeEstrelas);
```

# Supplier<T>

- Supplier é uma interface que é utilizada para gerar informação. Não tem parâmetros e devolve um resultado do tipo T.

```
@FunctionalInterface
public interface Supplier<T>
{
    T get();
}
```

- Pode ser utilizada por exemplo para permitir criar métodos que fazem pretty printing de informação.
- Criam-se expressões de pretty printing e aplicam-se a todos os objectos
- Com isto evita-se estar sempre a alterar o método toString.