

- 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");
}
```

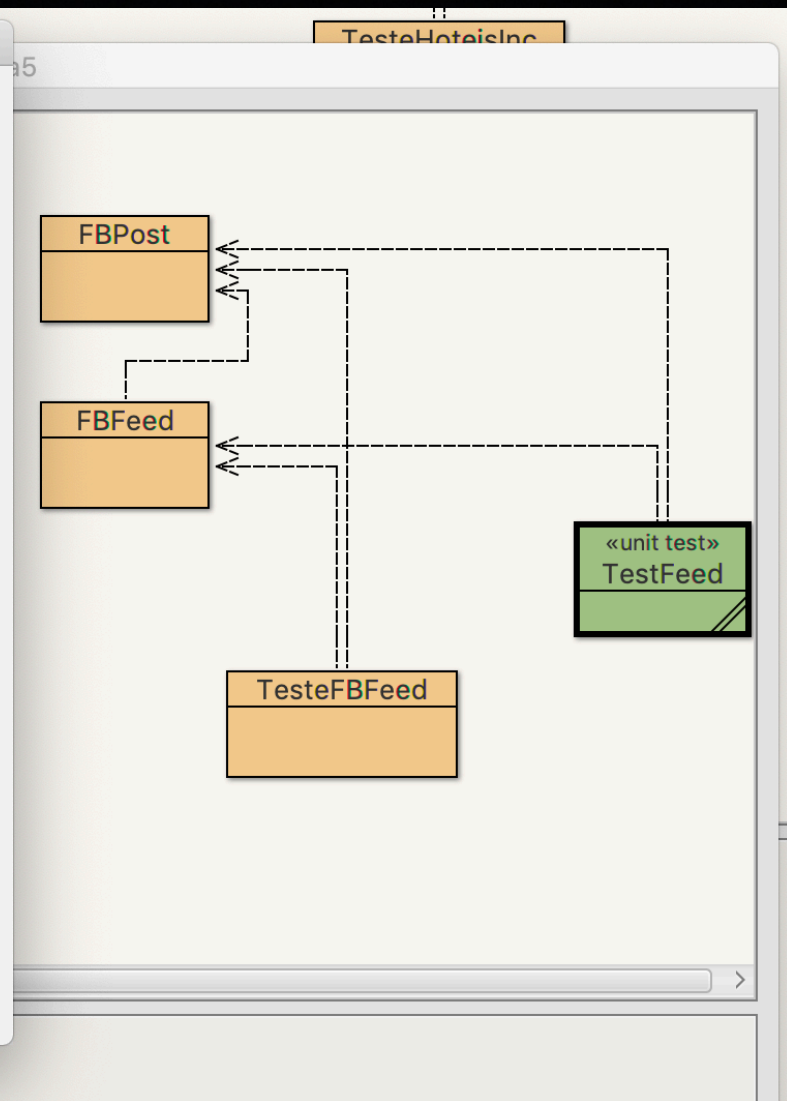


BlueJ: Test Results

- ✓ 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
  - carregar de ficheiro um novo defesa/atacante/etc. com um identificador igual a um já existente.



- 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).



- 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 *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
- garante-se assim que o surgimento de um erro obriga o programador a criar código para o tratar. Em vez de apenas o ignorar...

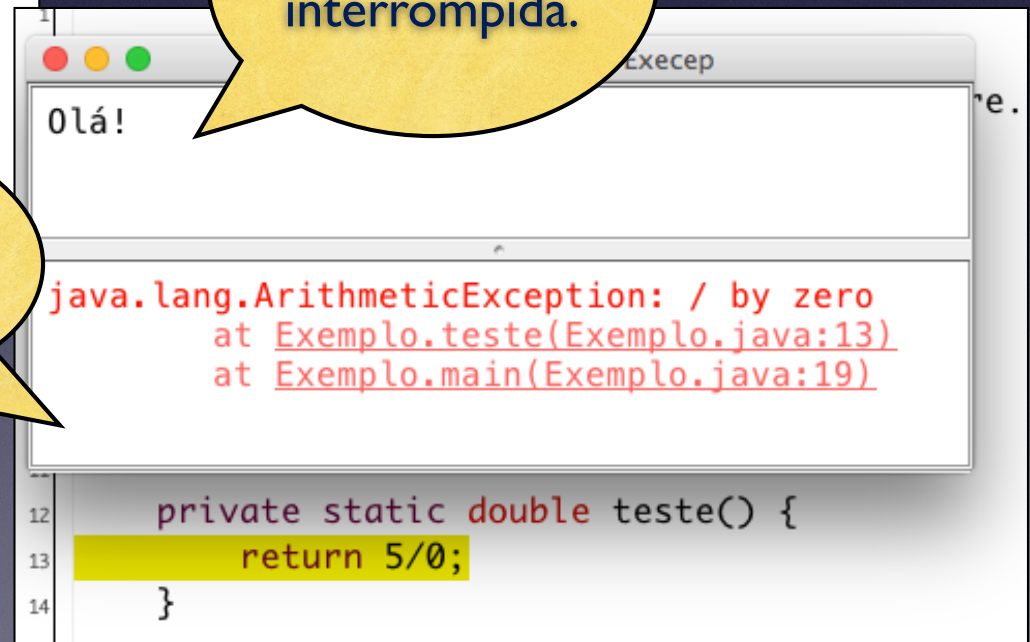


# Exceções

```
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.



```
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**.

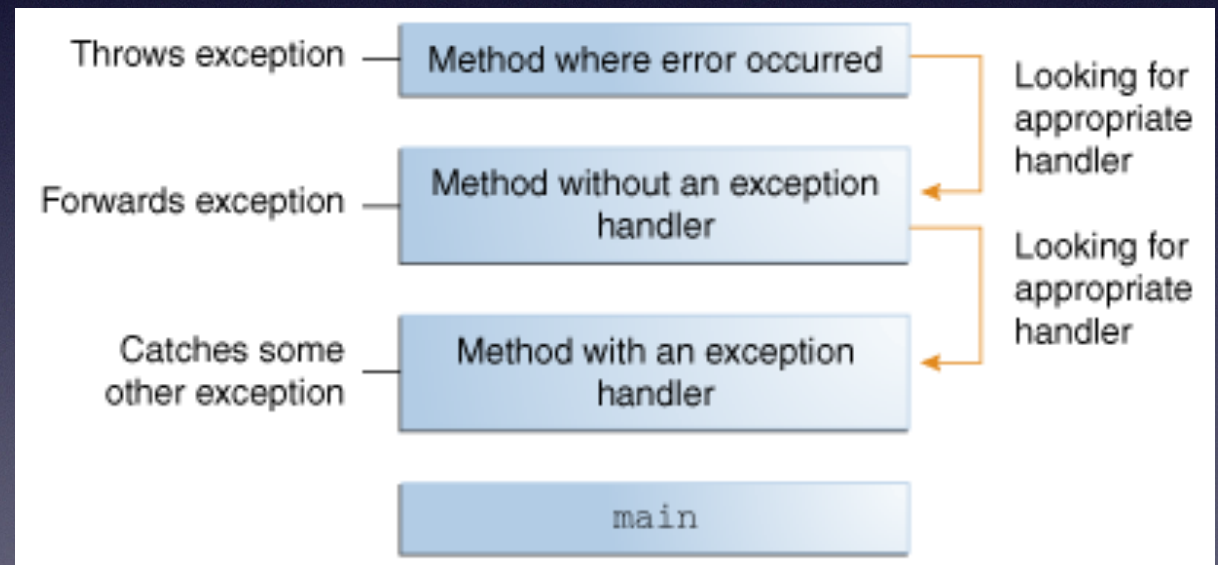
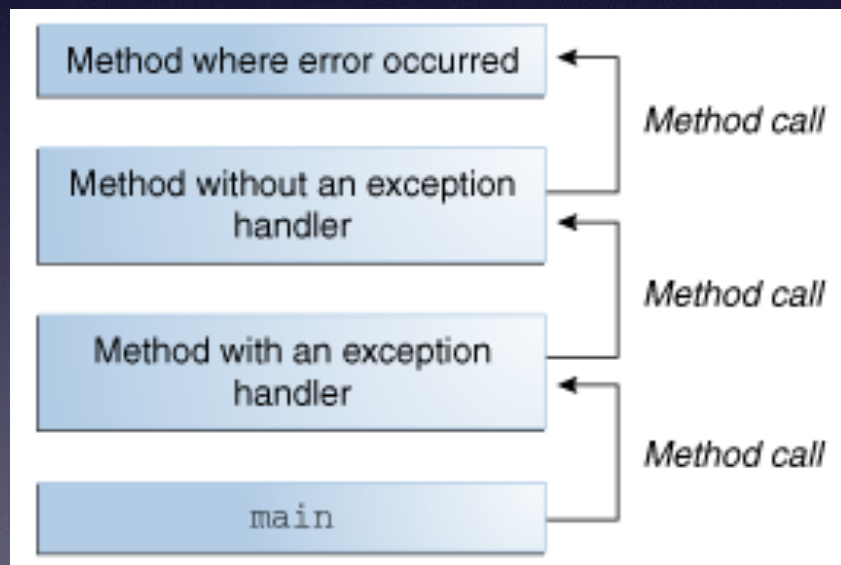
BlueJ: Terminal Window - Execep

```
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  
 * @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();  
}
```

Obrigatório  
declarar que lança  
exceção.

Lança uma  
exceção.

```
public static void main(String[] args) {  
  
    ...  
    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: **IOError**
- 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 exceções 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) {
```

```
    // carregar informação
```

```
    do {
```

```
        menumain.executa();
```

```
        switch (menu.getOpcao()) {
```

```
            case 1 // invocar método 1  
                break;
```

```
            case 2 // invocar método 2  
                break;
```

```
            case 3 // invocar método 3  
                break;
```

```
            case 4 // invocar método 4  
                break;
```

```
            case 5 // invocar método 5  
                break;
```

```
            case 6 // invocar método 6
```

```
        }
```

```
    } 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).



# 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 logs 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 void parse(){
    List<String> linhas = lerFicheiro("dados.csv");
    String[] linhaPartida;
    for (String linha : linhas) {
        linhaPartida = linha.split(":", 2);
        switch(linhaPartida[0]){
            case "Guarda-Redes":
                GuardaRedes j = parseGuardaRedes(linhaPartida[1]);
                System.out.println(j.toString());
                break;
            (...)
            default:
                System.out.println("Linha inválida.");
                break;
        }
    }
    System.out.println("done!");
}
```