

Interfaces

permitem relacionar, sob a figura de um novo tipo de dados, objectos de classes não relacionadas hierarquicamente

possibilita manter uma hierarquia de herança e ter uma outra hierarquia de tipos de dados

possibilitam que um mesmo objecto possa apresentar-se sob a forma de diferentes tipos de dados, exibindo comportamentos diferentes, consoante o que seja pretendido

permite esconder a natureza do objecto e fazer a sua *tipagem* de acordo com as necessidades do momento

permite limitar os métodos que, em determinado momento, podem ser invocados num objecto

(muito relevante) possibilitam que o código possa ser desenvolvido apenas com o recurso à interface e sem saber qual a implementação

principal razão para termos utilizado
`List<E>`, `Set<E>`, `Map<K,V>`

o programador apenas necessita saber o comportamento oferecido e pode construir os seus programas em função disso

As declarações constantes de uma interface constituem um contrato, isto é, especificam a forma do comportamento que as implementações oferecem

Por forma a fazer programas apenas precisamos de saber isso e ter bem documentado o que cada método faz.

Em função disso podemos fazer o programa e os programas de teste.

Nas aulas práticas já se apresentou uma forma de efectuar testes unitários, isto é fazer programas que fazem asserções sobre o comportamento exibido pelos programas com determinados dados.

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

TesteHotelInc

FBPost

FBFeed

TesteFBFeed

«unit test»
TestFeed

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, 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, 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

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 proprietário 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 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

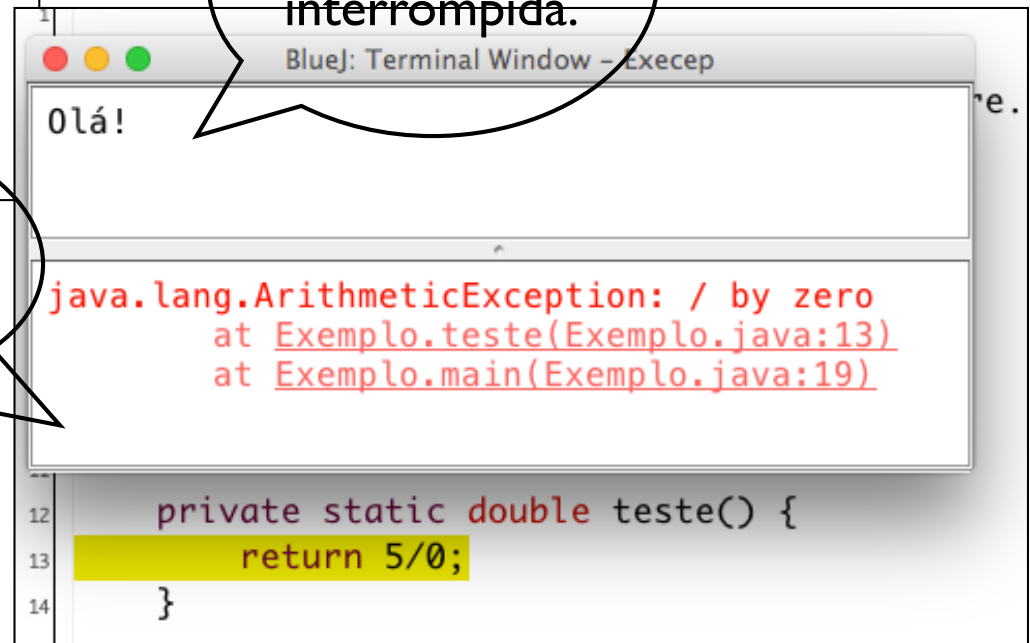
garante-se assim que o surgimento de um erro obriga o programador a criar código para o tratar. Em vez de apenas o ignorar...

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.

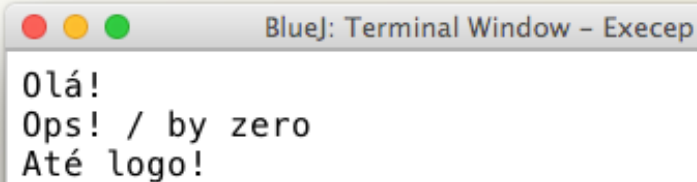


```
BlueJ: Terminal Window - Execep  
Olá!  
  
java.lang.ArithmeticException: / by zero  
    at Exemplo.teste(Exemplo.java:13)  
    at Exemplo.main(Exemplo.java:19)  
  
12     private static double teste() {  
13         return 5/0;  
14     }
```

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**.

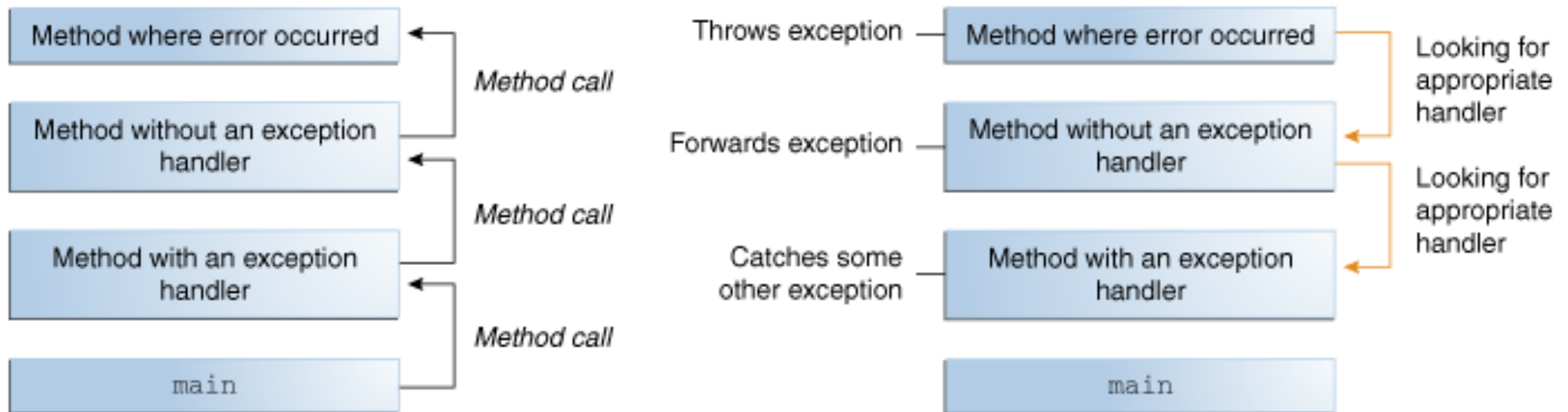


BlueJ: Terminal Window - Execep

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

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();  
}
```

Obrigatório
declarar que lança
exceção.

Lança uma
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: **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 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`

```
/**
 * 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;
}
```

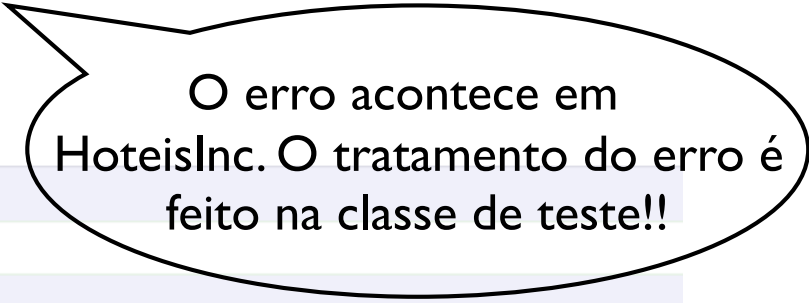
Utilização na classe de teste

```
//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: totalFolha(); // invocar método 4
                     break;
            case 5: totalPorTipo(); // invocar método 5
                     break;
            case 6: totalKms(); // invocar método 6
                     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!...");
}

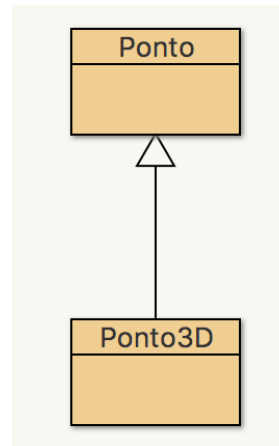
```

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

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

Tipos Parametrizados

Consideremos novamente a hierarquia dos pontos:



e considere-se que pretendemos manipular colecções de pontos

Como sabemos uma lista de pontos,
`List<Ponto>` pode conter instâncias de
`Ponto`, `Ponto3D` ou outras subclasses
destas classes.

no entanto, essa lista não é o supertipo das
listas de subtipos de `Ponto`!!

..., porque a hierarquia de `List<E>` não
tem a mesma estruturação da hierarquia de
`E`

Consideremos a classe ColPontos que tem uma lista de Ponto.

```
public class ColPontos {  
  
    private List<Ponto> meusPontos;  
  
    public ColPontos() {  
        this.meusPontos = new ArrayList<Ponto>();  
    }  
  
    public void setPontos(List<Ponto> pontos) {  
        this.meusPontos = pontos.stream().map(Ponto::clone).collect(Collectors.toList());  
    }  
}
```


Seja agora uma classe que utiliza uma instância de ColPontos

```
public class TesteColPontos {  
    public static void main(String[] args) {  
        Ponto3D r1 = new Ponto3D(1,1,1);  
        Ponto3D r2 = new Ponto3D(10,5,3);  
        Ponto3D r3 = new Ponto3D(4,16,7);
```

```
        ArrayList<Ponto3D> pontos = new ArrayList<>();  
        ColPontos colecao = new ColPontos();  
        colecao.setPontos(pontos);
```

List<Ponto3D>
não pode ser vista como
List<Ponto>!!

```
incompatible types:  
java.util.ArrayList<Ponto3D> cannot be  
converted to java.util.List<Ponto>
```

ass compiled - no syntax errors

O tipo das Lists de Ponto e das Lists suas subtipos declara-se como:

List<? extends Ponto>

O tipo das Lists de super-classes de Ponto declara-se como:

List<? super Ponto>

Será necessário alterar o código dos métodos para permitir esta compatibilidade de tipos:

```
public void setPontos(List<? extends Ponto> pontos) {  
    this.meusPontos = pontos.stream().map(Ponto::clone).collect(Collectors.toList());  
}
```

Collection<? extends Ponto> =
Collection<Ponto3D> ou
Collection<PontoComCor> ou ...

Ainda sobre ordenações

Temos visto que podemos ordenar
coleções de dados recorrendo:

à ordem natural, através do método
`compareTo()` (interface
`Comparable<T>`)

a uma relação de ordem a fornecer,
através do método `compare(...)`
(interface `Comparator<T>`)

Temos utilizado o `TreeSet<E>` como mecanismo base para fazer ordenações:

tirando partido de que o `TreeSet` utiliza uma relação de ordem para colocar os objectos

como o conjunto não admite repetidos é preciso especial cuidado com comparações que devolvem 0 (são iguais)

ou se acrescentam mais critérios de comparação ou então alguns elementos são ignorados

Uma outra forma é utilizar as `List<E>` para efectuar a ordenação, não tendo que ter de prever a situação repetição de dados

utilizando o método `List.sort(c)`

utilizando o `sorted()` e `sorted(c)`
das streams

com recurso a List.sort

```
/**
 * A estratégia de colocar comparators em TreeSet necessita que o método compare
 * , do Comparator, não dê como resultado zero (caso em que o Set não permite ficar
 * com elementos repetidos). Claro que isto nem sempre é possível...
 *
 * Uma forma de permitir continuar a ter repetições é assumir que a estrutura de
 * dados é uma lista e utilizar o método sort (método de classe de Collections)
 * passando como parâmetro um comparator.
 */

public List<Hotel> ordenarHoteisList(Comparator<Hotel> c) {
    List<Hotel> l = new ArrayList<Hotel>();
    hoteis.values().forEach(h -> {
        l.add(h.clone());
    });

    l.sort(c);
    return l;
}
```

com recurso a sorted

```
/**
 * Com a utilização de Stream pode efectuar-se directamente sobre a stream a
 * ordenação. Através do método sorted(), para a ordem natural, ou do método
 * sorted(c) para um comparador definido.
 */

public List<Hotel> ordenarHoteisListStream(Comparator<Hotel> c) {
    return this.hoteis.values().stream()
        .map(Hotel::clone).sorted(c).collect(Collectors.toList());
}
```