

# Tutorial 1 – Parte B

Uso do Depurador (*Debugger*) do Eclipse



Unidade Curricular de  
Laboratório de Programação

2020/2021

# Avaliação dos alunos

Como já viram no Moodle, nesta semana não terão que fazer um exercício adicional ao tutorial.

A vossa avaliação será através de *screenshots* que vocês irão fazendo ao longo do tutorial, que depois deverão entregar no Moodle, reunidos num zip chamado `DebugScreenshots.zip`.

Deverão estar atentos, durante o tutorial, para a figura na margem da página.



Esta figura chama a atenção para ações que devem fazer que são importantes para a avaliação.

## Porque devo usar o *Debugger*?

A melhor forma de detetarmos o que está a correr mal com o nosso programa é usarmos um *depurador* (em Inglês, *debugger*).

A tarefa de *depuração* de um programa (*debugging*) permite-nos, por exemplo, correr o programa passo a passo enquanto visualizamos o código e os valores que as suas variáveis vão tomando.

Podemos definir *pontos de paragem* (*breakpoints*) no código fonte para indicar locais no código onde a execução do programa deverá parar durante o processo de depuração. Nestes pontos poderemos inspecionar os valores das variáveis, alterar o seu conteúdo, etc.

Para parar a execução, se um campo for lido ou modificado, podemos especificar *pontos de visualização* (*watchpoints*).

O Eclipse tem incorporado um modo de *debug* para Java e fornece uma perspetiva de depuração com um conjunto pré-configurado de vistas e permite controlar o fluxo de execução através de comandos de depuração.

# Vamos então começar

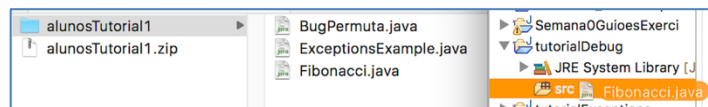
Se não o fez já, descarregue o ficheiro `alunosTutorial1.zip` acessível na página de LabP para o seu disco.

Descompacte o zip, obtendo a pasta `alunosTutorial1`.

No Eclipse, crie um projeto Java (**File**→**New**→**Java Project**).

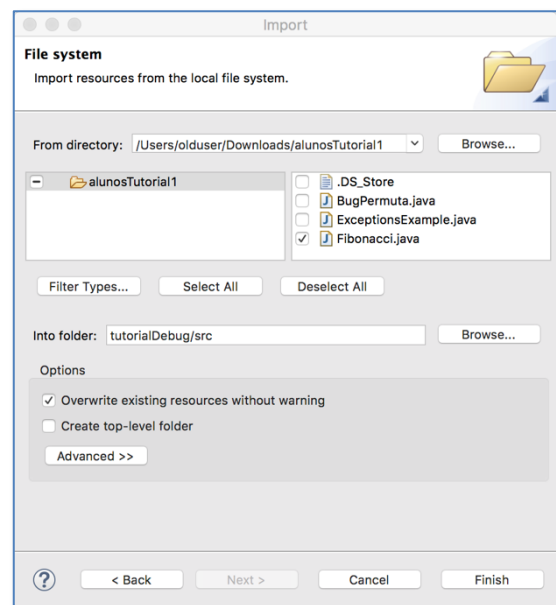
Agora vai adicionar-lhe a classe `Fibonacci` que está incluída naquela pasta. Pode fazer isto de várias formas:

- “arrastando” o ficheiro `Fibonacci.java`, no sistema de ficheiros, para cima da pasta `src` do novo projeto, no Eclipse;



**OU**

- Com o novo projeto selecionado, escolher **File**→**Import**
  - Na janela seguinte escolher **General/File System** e botão “Next”;
  - De seguida clicar em “Browse” no “From Directory” e selecionar a pasta `alunosTutorial1` e botão “Open”;
  - Nas caixas, selecionar `Fibonacci.java`
  - De seguida clicar em “Browse” no “Into folder” e selecionar a pasta `src` do projeto `tutorialDebug` e botão “Open”;
  - Clicar em “Finish”.



Verifique que já tem a classe `Fibonacci` na pasta `src` do seu novo projeto.

Clique duas vezes na classe para a abrir no editor do Eclipse.

Altere os nomes e número no `@author` para o seu próprio nome e número.



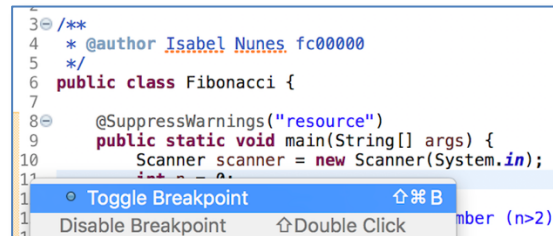
Vamos começar por fazer *debugging* do método `main`.

## BreakPoints

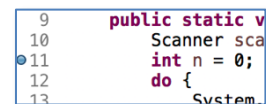
Um *breakpoint* é um local no código onde a execução será interrompida para que se possam analisar as informações do estado do programa.

Vamos adicionar um *breakpoint* na linha 11, onde é feita a declaração do `n`:

- clicar com o botão direito do rato sobre a barra lateral esquerda do editor Java na linha de código na qual se pretende que o programa pare. Escolher a opção “Toggle Breakpoint”. Obtemos o mesmo efeito clicando duas vezes no lado esquerdo do editor.



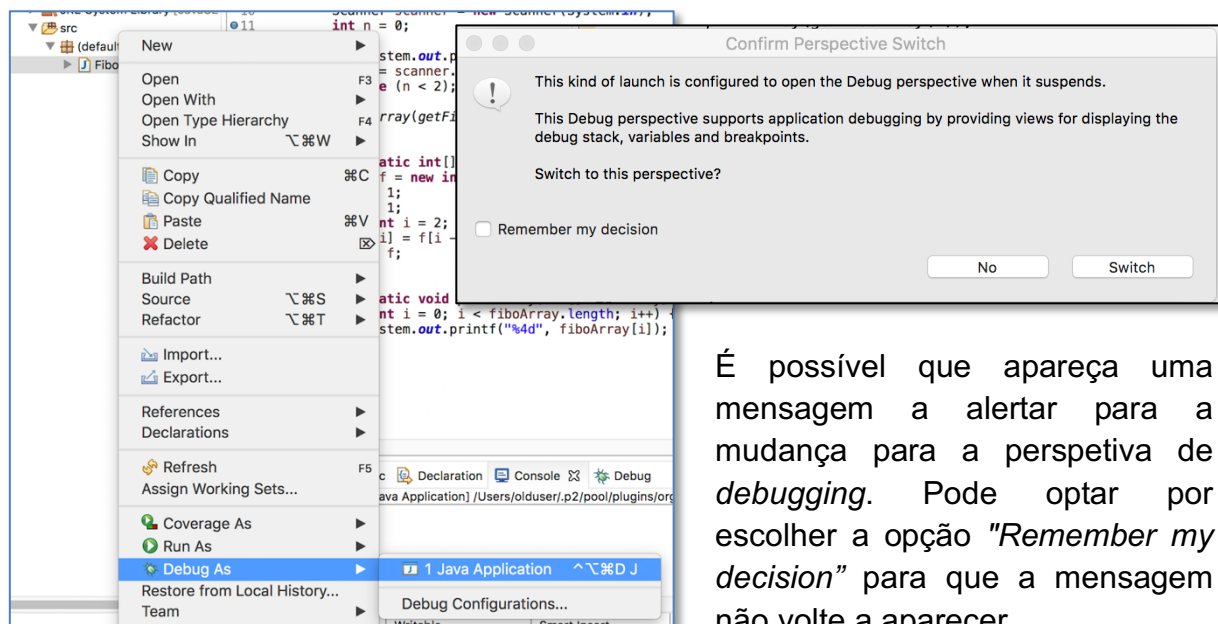
A indicação de que um *breakpoint* foi adicionado é um pequeno ponto azul que deverá aparecer ao lado da linha de código onde a execução será interrompida.



## Executar o programa através do debugger

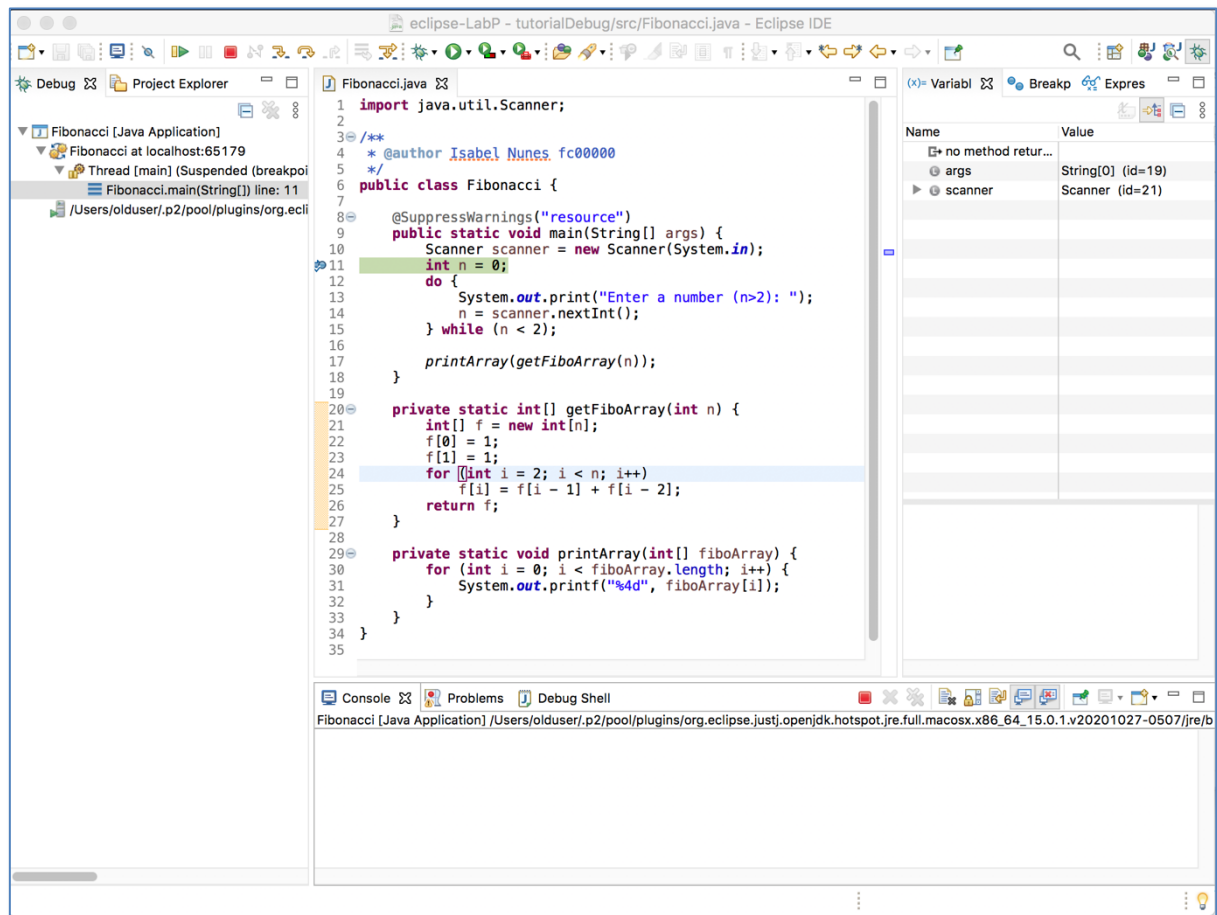
Clicar com o botão direito do rato sobre o ficheiro `Fibonacci.java` e seleccionar

**Debug As → Java Application** (ver figura seguinte) ou clicar no símbolo .



É possível que apareça uma mensagem a alertar para a mudança para a perspetiva de *debugging*. Pode optar por escolher a opção “*Remember my decision*” para que a mensagem não volte a aparecer.

Clicando em “Switch” aparece algo semelhante à figura seguinte, uma representação da perspectiva de *debugging*.



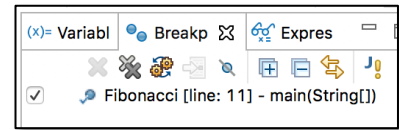
Faça agora o seu primeiro *screenshot* do ambiente Eclipse, com atenção para apanhar a linha do @author, que já deverá conter o seu nome e número. Dê o nome debugScreenshot1 ao seu ficheiro.



Vamos agora analisar as vistas desta perspectiva.

- **Debug:** mostra (em *real time*) o traço da pilha de execução dos métodos que estão a ser usados. No exemplo apenas aparece o método `main` do programa `Fibonacci`. Se outros programas estivessem em execução, estes seriam vistos nesta parte da perspectiva.

- **Variables:** mostra os valores de todas as variáveis que foram declaradas até aqui e que ainda têm alguma referência do programa em execução. Como os *breakpoints* param a execução na linha imediatamente anterior, o *n* ainda não está declarado e por isso não aparece.
- **Breakpoints:** mostra a localização de todos os *breakpoints* que foram introduzidos no código.
- **Fibonacci.java:** apresenta o código no momento do *breakpoint*.



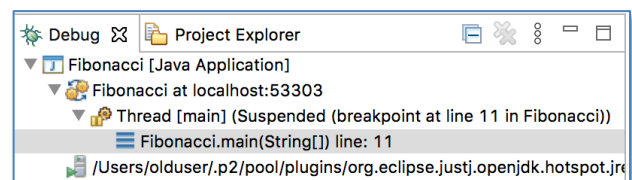
Observe os novos comandos de *debug* na parte de cima da janela:



Teclas	Descrição
F5 <i>Step into</i>	Executa a linha selecionada atualmente e vai para a próxima linha do programa. Se a linha selecionada é uma chamada de um método, o <i>debugger</i> entra no código associado a esse método.
F6 <i>Step over</i>	Passa sobre a chamada, isto é, executa um método sem entrar no método.
F7 <i>Step return</i>	Termina o método em execução saltando para a linha seguinte do método que o chamou.
F8 <i>Resume</i>	Informa o <i>debugger</i> do Eclipse para retomar a execução do código do programa até atingir o próximo <i>breakpoint</i> ou <i>watchpoint</i> ou até o programa terminar.

Sempre que deseje interromper a depuração do programa pode carregar no botão

Voltando à execução do nosso programa, pode observar na vista **Debug** que o programa (aqui chamado de “*Thread*”) está suspenso, devido ao *breakpoint* na linha 11.



Vamos continuar.

Pressione agora o botão **Step Into** (ou **F5**) ou o botão **Step Over** (ou **F6**) para que a próxima linha seja executada.

Após a execução da linha 11, observe agora na janela **Variables** que a nova variável `n` foi declarada e tem o valor zero.

Name	Value
args	String[0] (id=19)
scanner	Scanner (id=20)
n	0

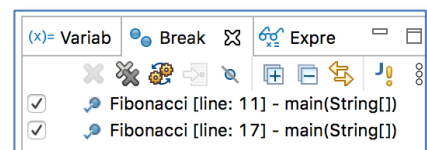
```

6 public class Fibonacci {
7
8     @SuppressWarnings("resource")
9     public static void main(String[] args) {
10         Scanner scanner = new Scanner(System.in);
11         int n = 0;
12         do {
13             System.out.print("Enter a number: ");
14             n = scanner.nextInt();
15             while (n < 2);
16         } while (n < 2);
17         printArray(getFiboArray(n));
18     }
19 }

```

Observe também que a execução já está dentro do ciclo `do-while` e que o programa está suspenso na linha que se segue à 11 (`System.out.println(...)`).

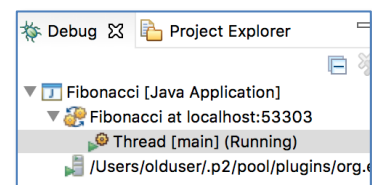
Antes de continuar, adicione outro ponto de interrupção na chamada ao método `printArray` (linha 17) e verifique na vista **Breakpoints** que já tem já lá está representado esse novo *breakpoint*.



Agora vamos deixar que o programa seja executado até chegar ao próximo *breakpoint*. Para isso clique em **Resume**  ou em **F8**.

Note que o programa não terminou, mas não está a fazer nada. Isto é porque o programa, depois de executar a instrução 13 (pode ver o *output* na consola), está agora à espera que o utilizador introduza um inteiro pelo teclado.

Podemos ver de novo na vista **Debug** que o programa não está suspenso, mas sim a executar (agora indica que está *Running*, e não *Suspended*, como antes).



```

Fibonacci [Java Application] /User
Enter a number (n>2): 13

```

Na **Console**, digite o número 13 e pressione *Enter*.

Agora a execução é suspendida imediatamente antes da chamada dos métodos `getFiboArray` e `printArray`.


```


16
17     printArray(getFiboArray(n));
18 }

```

## Passando por vários métodos

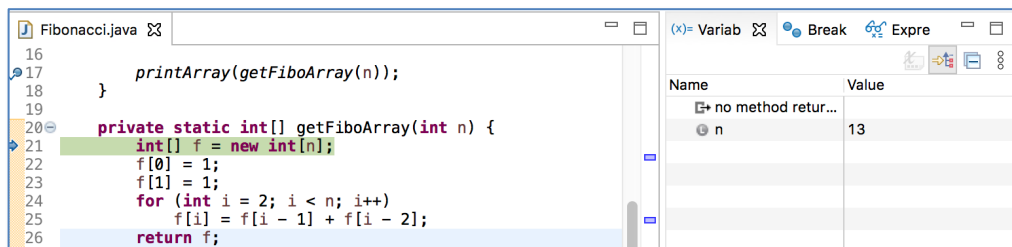
Agora que passámos pela maior parte do método `main`, vamos entrar noutros métodos. A linha atual em que o programa está suspenso irá executar em primeiro lugar o método `getFiboArray` e só depois o método `printArray`.


Não o vamos fazer, mas se clicássemos agora em **Step Over**  estes métodos seriam ignorados em termos de *debugging*, embora fossem executados, claro.

Mas interessa-nos acompanhar a evolução dos valores através da execução desses métodos, por isso clicamos o botão **Step Into** . A imagem ao lado ilustra o que se obtém.

```
19
20 private static int[] getFiboArray(int n) {
21     int[] f = new int[n];
22     f[0] = 1;
23     f[1] = 1;
24     for (int i = 2; i < n; i++)
25         f[i] = f[i - 1] + f[i - 2];
26     return f;
27 }
```

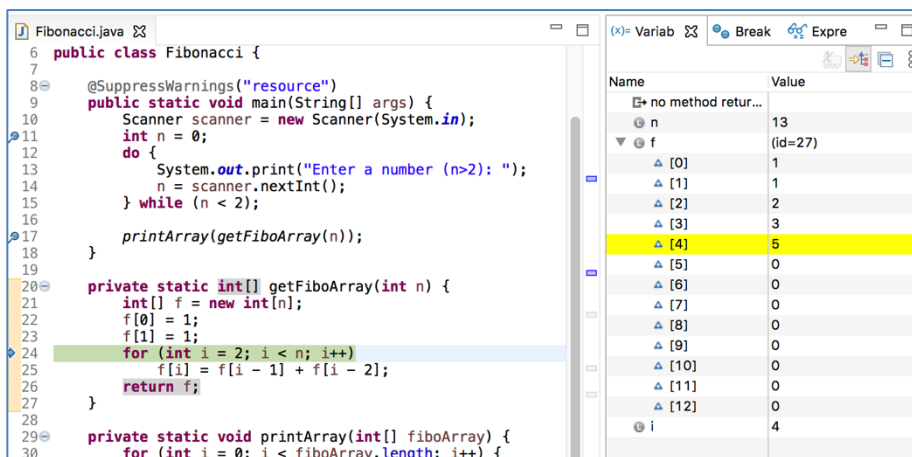
Observe ainda que na vista **Variables** são mostradas novas variáveis (neste momento é mostrado somente o parâmetro `n` do método `getFiboArray` com o valor 13, pois a instrução 21 ainda não foi executada).



Se carregar outra vez em **Step Into** , verá que aparece a variável `f` também. Como `f` é um *array*, se carregar no triângulozinho à sua esquerda verá o conteúdo de cada um dos seus 13 elementos (agora ainda estão todos a zero).

n	13
f	(id=27)

Se carregar mais vezes em **Step Into** , verá o conteúdo dos elementos de `f` a



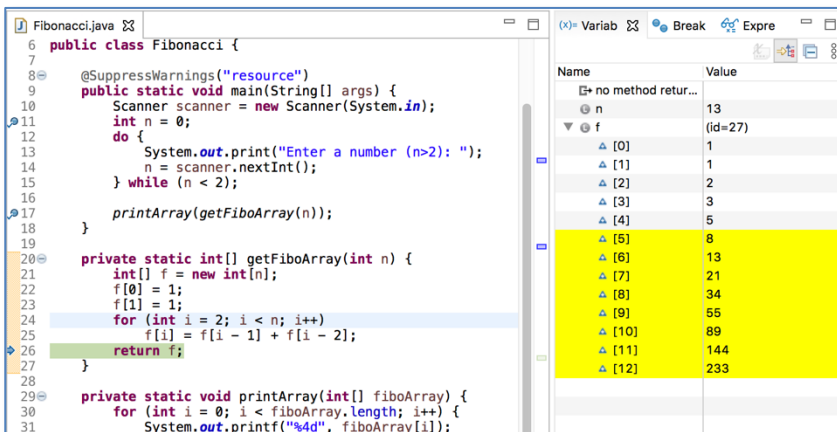
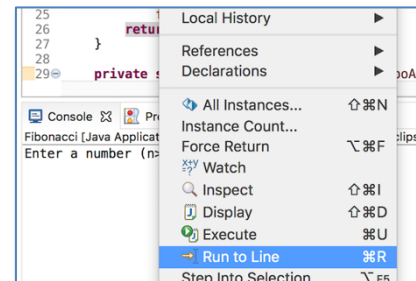
mudar, ao mesmo tempo que vê quais as instruções que vão sendo executadas. Repare que também aparece a variável de progresso do ciclo (`i`).



Estas são as variáveis que estão atualmente no âmbito do método (recorde de IP o que aprendeu sobre o âmbito das variáveis). Note ainda que o âmbito das variáveis é local e portanto as variáveis declaradas no método `main` não estão neste contexto e por essa razão não estão visíveis.

Vamos ver uma outra maneira de percorrer um programa sem usar *breakpoints* e sem ter que o percorrer linha a linha.

Clicando com o botão direito do rato na linha 26 (`return f;`), escolher a opção **Run to Line**, que funciona como se fosse um *breakpoint* inserido nesta linha.



A imagem ao lado mostra o conteúdo das variáveis após esta ação. Pode ver que todos os elementos de `f` foram preenchidos e que a variável `i` desapareceu (pois era local ao ciclo `for`, que entretanto terminou).

Faça agora o seu segundo *screenshot* do ambiente Eclipse, com atenção para que a linha do `@author` também apareça. Dê o nome `debugScreenshot2` ao seu ficheiro.



Continuando...

Não vamos fazer isso, mas bastaria agora voltar a carregar em **Step Into** (🔍) para que o fluxo de execução retornasse à linha do método `main` que invocou o método, pois a instrução `return f` seria executada.

Vamos antes usar o botão **Step Return** (🔍) ou pressionar **F7** que é o que se costuma usar sempre que queremos executar todas as instruções que faltam de um método,


de modo a que o fluxo de execução retorne à linha que o invocou. A execução volta então à linha 17.

```
15         write(n < 2),
16
17         printArray(getFiboArray(n));
18     }
19 }
```

```
28
29 private static void printArray(int[] fiboArray) {
30     for (int i = 0; i < fiboArray.length; i++) {
31         System.out.printf("%4d", fiboArray[i]);
32     }
33 }
```

Ao clicar novamente em **Step Into** irá navegar para o método `printArray`.



Verifique de novo na vista **Variables** o conteúdo do parâmetro `fiboArray`.

Se clicar agora em **Resume**  o programa executará até terminar pois já não existem mais *breakpoints* onde parar.

## Breakpoints especiais

As linhas não são o único sítio onde podemos especificar *breakpoints*. Podemos, por exemplo, fazer o programa parar sempre que ocorre uma exceção.

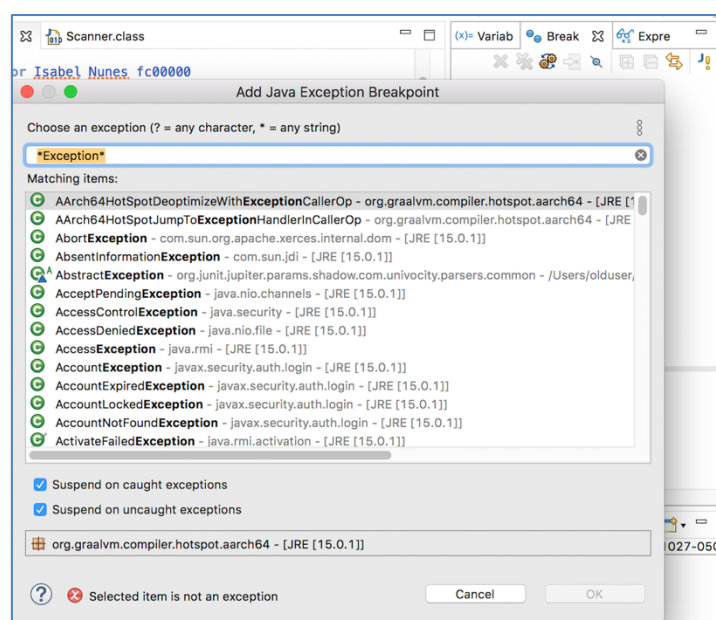
Vamos fazer isso mas primeiro vamos remover os *breakpoints* que já temos:

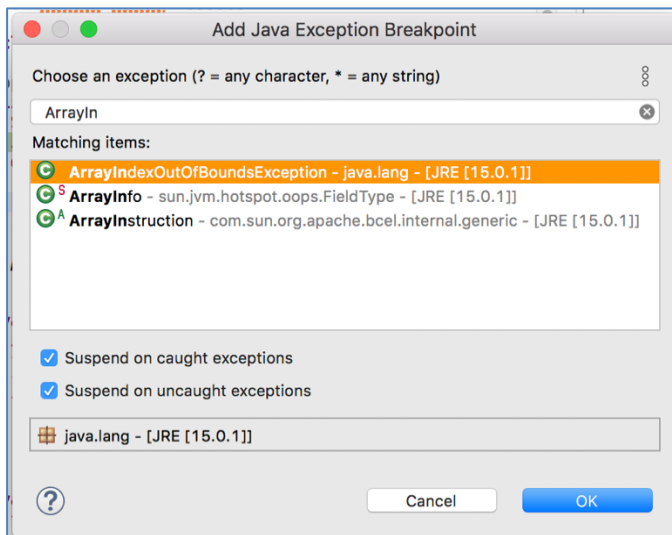
- Certifique-se de que o programa não está a ser executado (se ainda estiver, pode sempre clicar em );
- Na Perspetiva de *Debug*, clique na vista **Breakpoints** e, em seguida, clique em **Remove All Breakpoints** () e confirme que quer removê-los).

Vamos agora adicionar um *breakpoint* que obrigue a parar a execução sempre que ocorre uma exceção do tipo `ArrayIndexOutOfBoundsException`.

Na vista de **Breakpoints** clique no botão **Add Java Exception Breakpoint** (.

Aparecerá uma caixa de diálogo para selecionar o tipo de exceção.



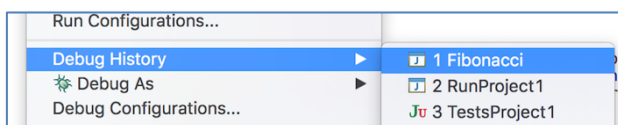



Por omissão, está definido que o programa termina com qualquer exceção.

Procure, digitando `ArrayIndexOutOfBoundsException` e clique em **OK** quando selecionar `ArrayIndexOutOfBoundsException` do pacote `java.lang`.

Vamos modificar o nosso código para provocarmos uma exceção deste tipo. No método `getFiboArray`, altere o sinal "`<`" para um "`<=`". Isso significa que o ciclo irá aceder a um elemento do vetor `fibo` que não existe.

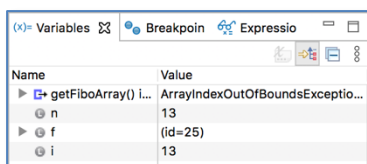
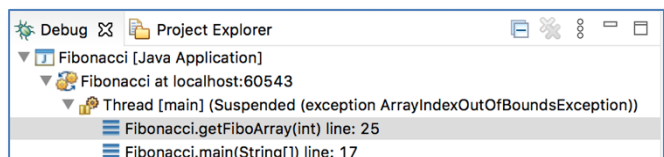
```
20 private static int[] getFiboArray(int n) {
21     int[] f = new int[n];
22     f[0] = 1;
23     f[1] = 1;
24     for (int i = 2; i <= n; i++)
25         f[i] = f[i - 1] + f[i - 2];
26     return f;
27 }
```



Grave o ficheiro e volte a executar o programa Fibonacci com o *debugger* (utilize o menu **Run**→**Debug History** →**Fibonacci** ou ).

Digite o número 13 quando solicitado.

Se desmarcou todos os *breakpoints* das secções anteriores, o *debugger* será interrompido quando a exceção for lançada. Saberá que ocorreu uma exceção porque é exibida na vista de **Debug** uma mensagem de suspensão.



Este modo é útil pois permite ver os valores de todas as variáveis aquando do lançamento de uma exceção.

Faça agora o seu terceiro *screenshot* do ambiente Eclipse (sempre com atenção para que a linha do `@author` também apareça). Dê o nome `debugScreenshot3` ao seu ficheiro.



Clique em **Resume**  para terminar o programa.

## Observar Expressões

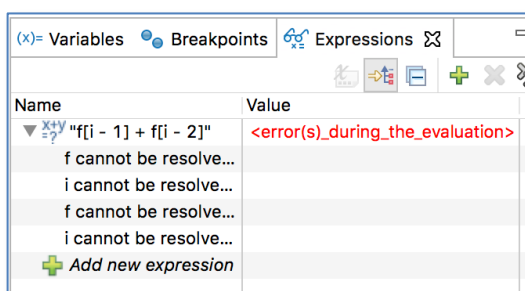
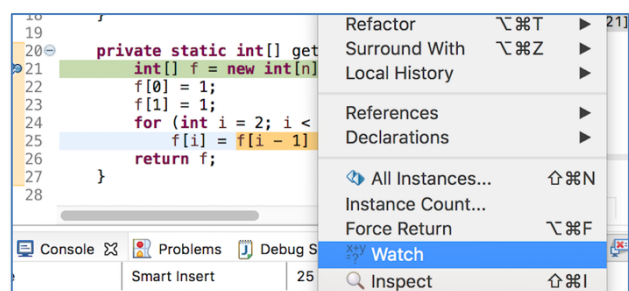
Além de manter o controlo sobre variáveis com o *debugger*, podemos também acompanhar a avaliação de expressões particulares em todo o programa usando o **Watch Expression**.

Para experimentar esta funcionalidade, vai primeiro:

- corrigir o bug injetado na secção anterior;
- adicionar um *breakpoint* na primeira linha do método `getFiboArray` (linha 21);
- iniciar de novo o programa em modo de depuração.


Digite 13 quando lhe for novamente pedido, para chegar ao *breakpoint* no método `getFiboArray`.

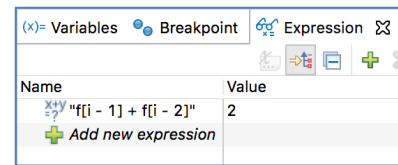
Agora selecione a expressão `f[i - 1] + f[i - 2]` e, clicando com o botão direito sobre ela, selecione **Watch**.



Note que esta expressão aparece agora na vista **Expressions**. Um erro é detetado na sua avaliação.

Expandindo a árvore vê-se que nem `f` nem `i` foram ainda inicializadas, uma vez que a execução foi suspensa antes de estas variáveis serem declaradas.

Utilize o **Step Over**  para percorrer o método. Até entrar no ciclo, onde a variável `i` é declarada, a expressão não é avaliada. Verifique que, logo que o fluxo de execução entra no ciclo, o valor da expressão é atualizado.



Esta é uma ferramenta muito útil para manter o controlo de expressões complexas.

Faça agora o seu quarto *screenshot* do ambiente Eclipse (sempre com atenção para que a linha do `@author` também apareça). Dê o nome `debugScreenshot4` ao seu ficheiro.



## Pratique agora o que aprendeu

Acrescente o ficheiro `BugPermuta.java` contido na pasta `alunosTutorial1` ao seu projeto.

Elimine todos os erros que encontrar no programa, usando as funcionalidades de *debugging* do Eclipse.

Espera-se que, com todas as correções feitas, no fim imprima uma permutação de um vetor que é lido.

## Entrega para avaliação:

Os 4 ficheiros com os *screenshots* que fez durante este tutorial devem ser colocados numa pasta de nome `DebugScreenshots`.

Um *zip* desta pasta deve ser entregue no Moodle, juntamente com os *screenshots* do tutorial de exceções.

Este guião é uma tradução livre e adaptada dos guiões:

<http://agile.csc.ncsu.edu/SEMaterials/tutorials/eclipse-debugger/>  
(não mais disponível online)

<http://www.vogella.com/tutorials/EclipseDebugging/article.html>

e

[https://www.eclipse.org/community/eclipse\\_newsletter/2017/june/article1.php](https://www.eclipse.org/community/eclipse_newsletter/2017/june/article1.php)

Todos os créditos são devidos aos seus autores.