CAPÍTULO 22

Apêndice - Debugging

"Olho por olho, e o mundo acabará cego."
— Mohandas Gandhi

22.1 - O QUE É DEBUGAR

Debugging (em português, depuração ou depurar) é um processo de reduzir ou encontrar bugs no seu sistema. De uma forma geral, debugging não é uma tarefa fácil de ser executada. Muitas variações podem atrapalhar esse processo, por exemplo, a linguagem que estamos utilizando e ferramentas disponíveis para fazermos debugging de um código.

O Java em si facilita muito neste processo, pois nos fornece maneiras de sabermos se o código está errado, por exemplo as exceptions. Em linguagens de baixo nível saber onde o bug estava era extremamente complicado. O que também facilita nosso trabalho são as ferramentas de debug. Veremos que elas são necessárias nos casos que nossos testes de unidade de logging não foram suficientes para encontrar a razão de um problema.

22.2 - DEBUGANDO NO ECLIPSE

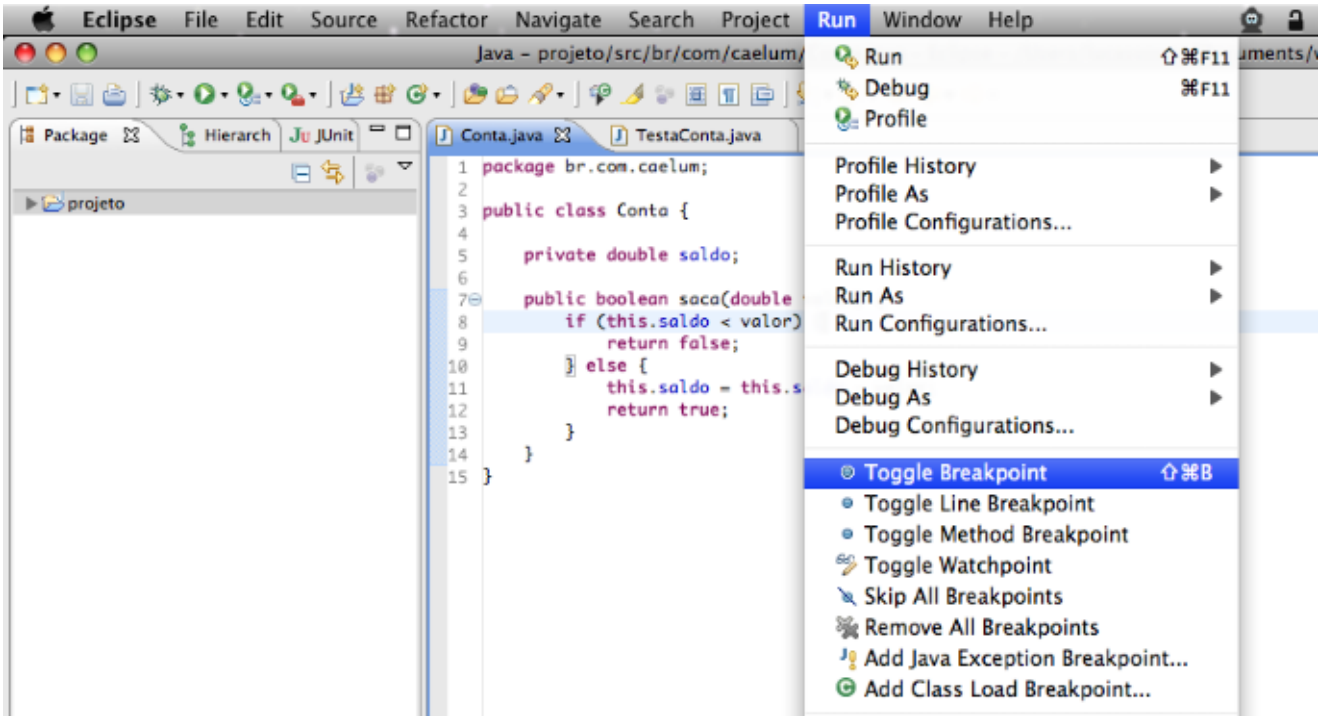
No curso utilizamos o Eclipse como IDE para desenvolvermos nosso código. Como foi dito ferramentas de debugging facilitam muito nosso trabalho, o Eclipse é uma das IDEs mais poderosas do mercado e nos fornece uma ferramenta que torna o processo extremamente simples.

O primeiro recurso que temos que conhecer quando começamos a debugar no Eclipse são os **breakpoints**. Eles são pontos de partida em nosso código para iniciarmos o processo de debug. Por exemplo, no código abaixo, imagine que desejamos debugar o comportamento do método `saca` da classe `Conta`, mais

especificamente do `if` que verifica se saldo é menor que o valor a ser sacado. Colocaríamos o **breakpoint** exatamente na linha `if (this.saldo < valor) {`:

```
public class Conta {

  private double saldo;

  public boolean saca(double valor) {
    if (this.saldo < valor) {
      return false;
    } else {
      this.saldo = this.saldo - valor;
      return true;
    }
  }
}
```

Mas como faço isso? Muito simples, basta clicar na linha que deseja adicionar o breakpoint, depois clicar no menu **Run -> Toogle Breakpoint**.



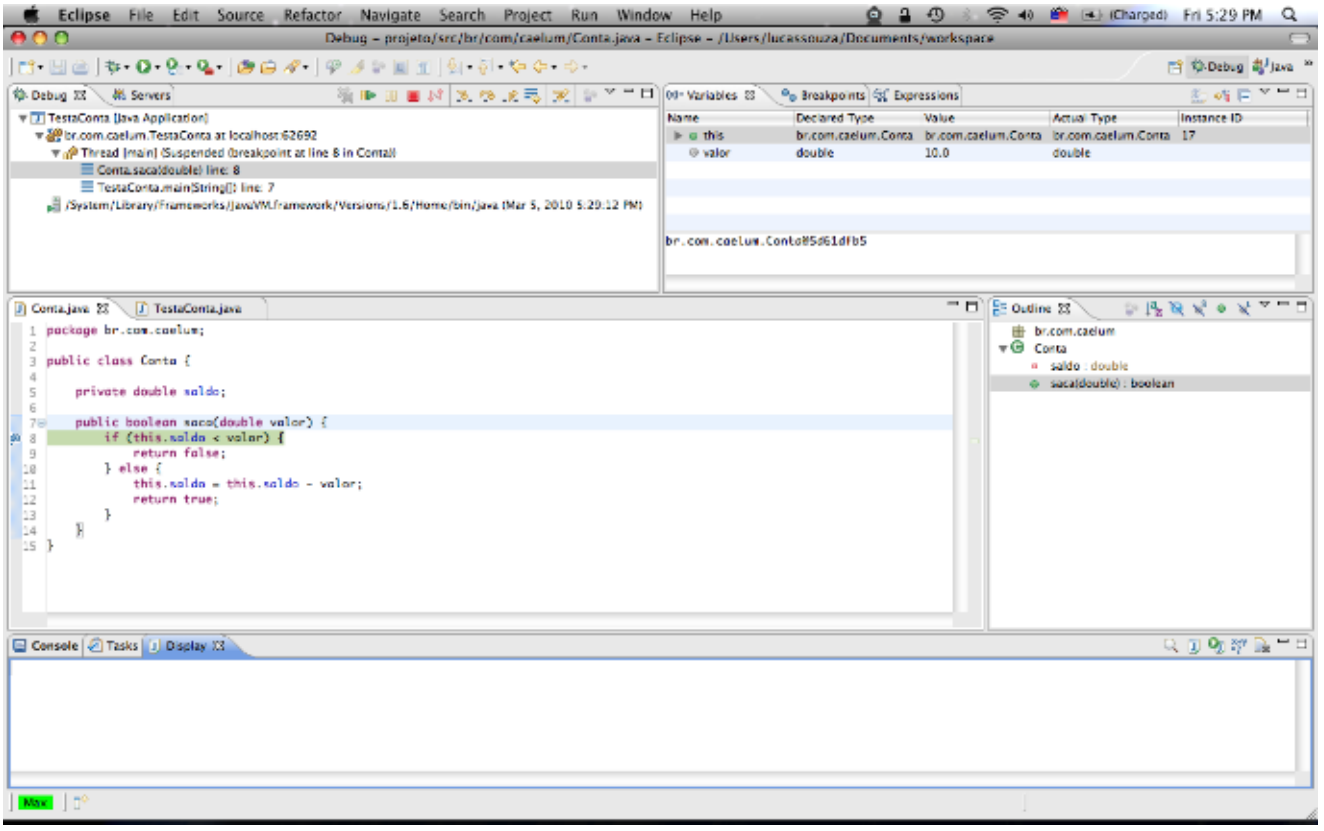
Esse é o tipo mais clássico de breakpoint, veremos alguns outros ao longo do capítulo.

Agora que já adicionamos o breakpoint que é o ponto de partida, vamos debugar nosso código. Precisamos rodar nosso código, ou seja, chamar o método `saca` para que o breakpoint seja encontrado. Teremos um código similar ao seguinte:

```
public class TestaConta {
  public static void main(String[] args) {
    Conta conta = new Conta();
    conta.saca(200);
  }
```

```
}
```

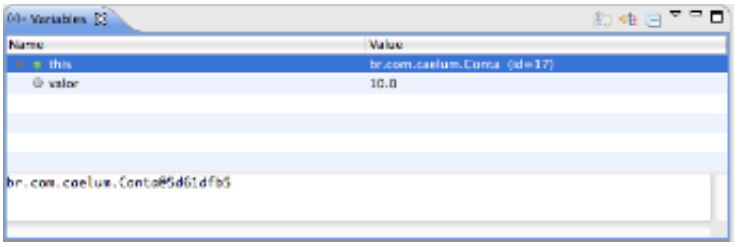
O processo normal para executarmos esse código seria clicar no menu *Run -> Run As -> Java Application*. Porém para rodar o nosso código em **modo debug** e ativar nosso breakpoint, devemos rodar o codigo no menu *Run -> Debug As -> Java Application*. Quando um breakpoint for encontrado no código que está sendo executado, o eclipse exibirá uma perspectiva específica de debug, apontando para a linha que tem o breakpoint.



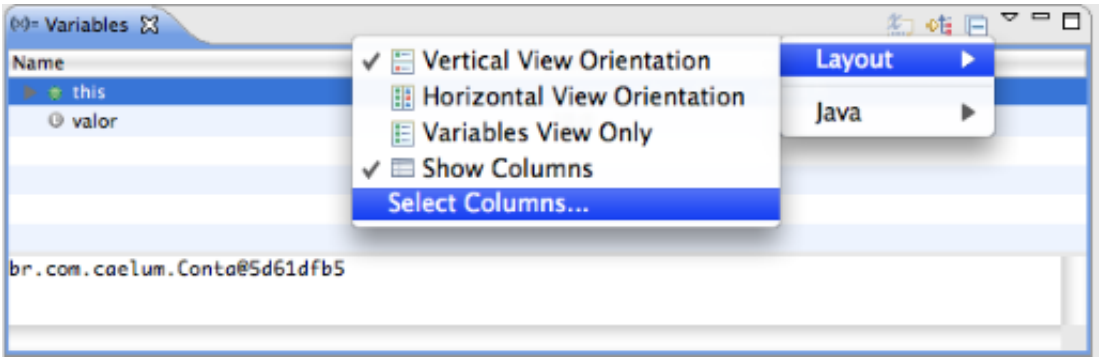
22.3 - Perspectiva de debug

Temos várias informações disponíveis nessa perspectiva, algumas são essenciais e básicas para trabalharmos com debug no nosso dia-a-dia, outras não tão relevantes e só usamos em casos muito específicos.

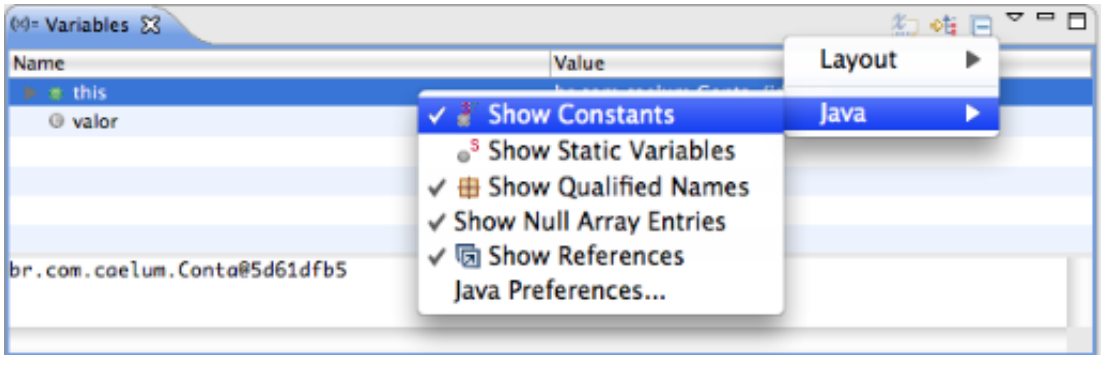
Dentro da perspectiva de debug, temos uma aba chamada `Variables`. São exibidas todas as variáveis encontradas dentro do código que você está debugando. Por exemplo, no debug que fizemos serão exibidas as variáveis do método `saca`, neste caso, `valor`. Além dos atributos de instância do objeto.



Podemos exibir mais informações sobre as variáveis, basta adicionarmos as colunas que desejamos na tabela exibida.



É possível também adicionarmos constantes e variáveis estáticas da classe que está sendo debugada.

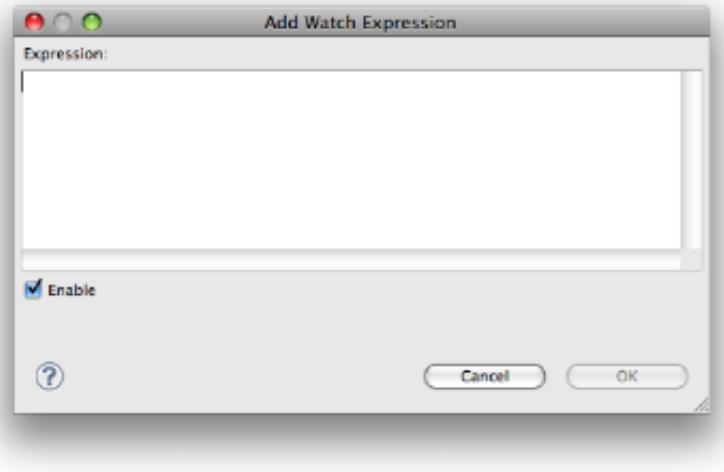


Na aba `Breakpoints` são exibidos todos os breakpoints que seu workspace possui. Mas por que isso é importante? É importante porque podemos ver todos os pontos de debug presentes e melhor, podemos desabilitá-los um a um ou todos de uma só vez. Você pode até mesmo pedir para exportar os breakpoints.

Para desabilitar ou habilitar todos breakpoints basta clicarmos no ícone **Skip**

All Breakpoints. Se quisermos desabilitar um a um, basta desmarcar o checkbox e o breakpoint será desativado. Às vezes, encontrar o código onde o breakpoint foi colocado pode ser complicado, na aba `Breakpoints` isso fica bem fácil de fazer, basta dar um duplo clique no breakpoint e o eclipse automaticamente nos mostra a classe "dona" dele.

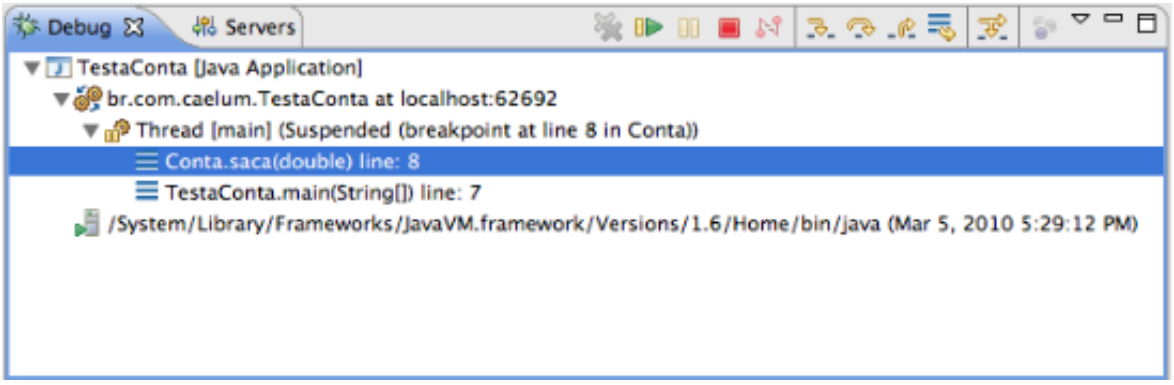
Quando estamos debugando código, muitas vezes é interessante saber o valor de alguma expressão ou método. Por exemplo, uma condição dentro de um if, `this.saldo > valor`. Esse valor não está em uma variável, ele está em uma expressão, o que pode tornar saber o valor dela complicado. A feature de `Expressions` descomplica esse processo para nós. Na perspectiva de Debug temos a aba `Expressions`. Basta clicar com o direito dentro da aba, e clicar em **Add Expression**:



E o resultado da expressão é exibido.

Temos outra aba importante chamada de `Debug`. Dentre as funções dela estão:

- **Threads** - Exibe as threads que estão sendo executadas, e melhor, mostra qual thread efetuou a chamada para o método onde está o debug. Além disso mostra a pilha de execução, o que nos permite voltar a chamada de um método
- **Barra de navegação** - Que permite alterarmos os caminhos que o debug seguirá.



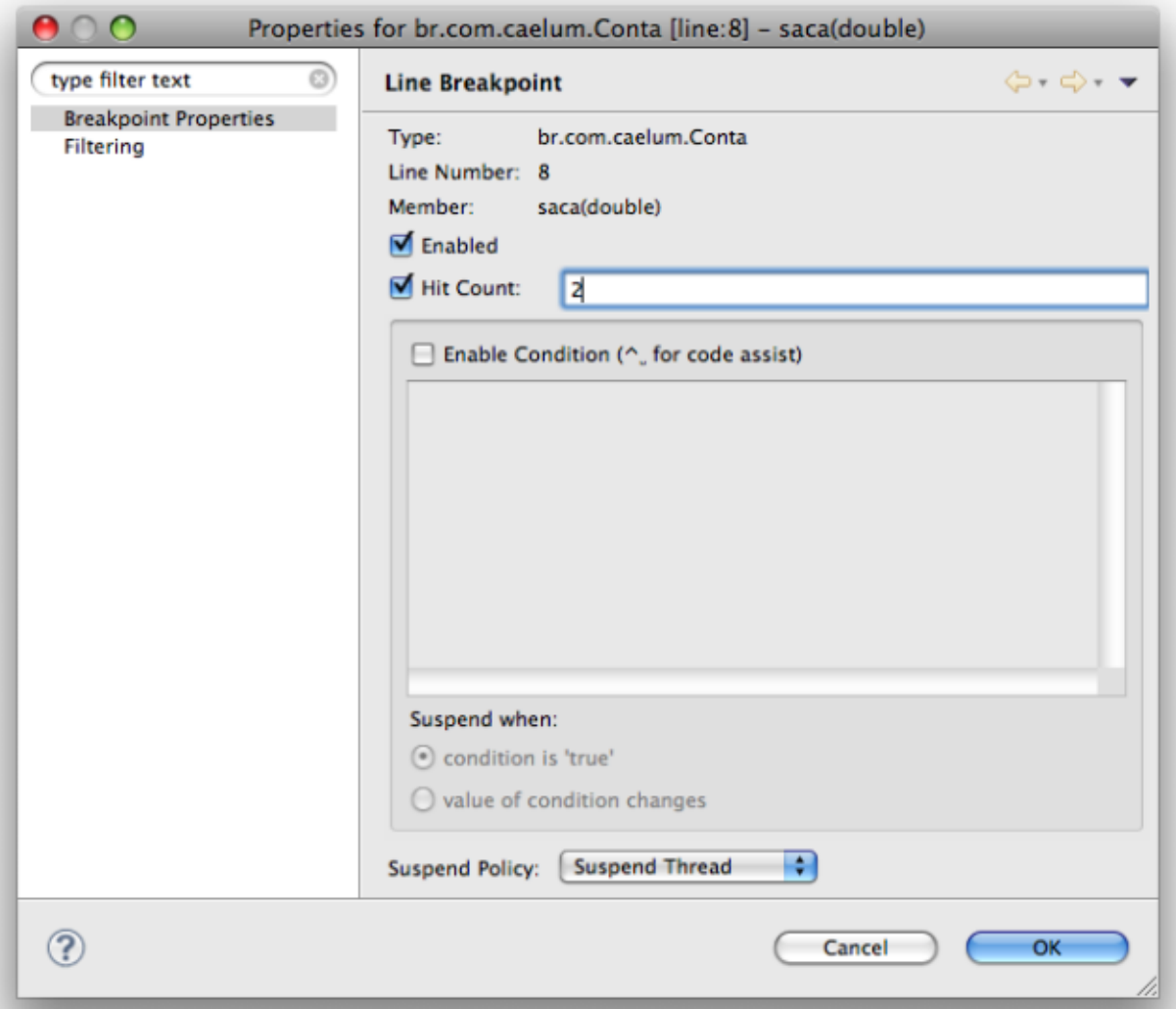
A lista a seguir mostrar algumas teclas e botões que alteram o caminho natural dos nosss debug:

- **F5** - Vai para o próximo passo do seu programa. Se o próximo passo for um método, ele entrará no código associado;
- **F6** - Também vai para o próximo passo, porém se o próximo passo for um método, ele não entrará no código associado;
- **F7** - Voltará e mostrará o método que fez a chamada para o código que está sendo debugado. No nosso caso voltará para o método `main` da classe `TestaConta`;
- **F8** - Vai para o próximo breakpoint, se nenhum for encontrado, o programa seguirá seu fluxo de execução normal.

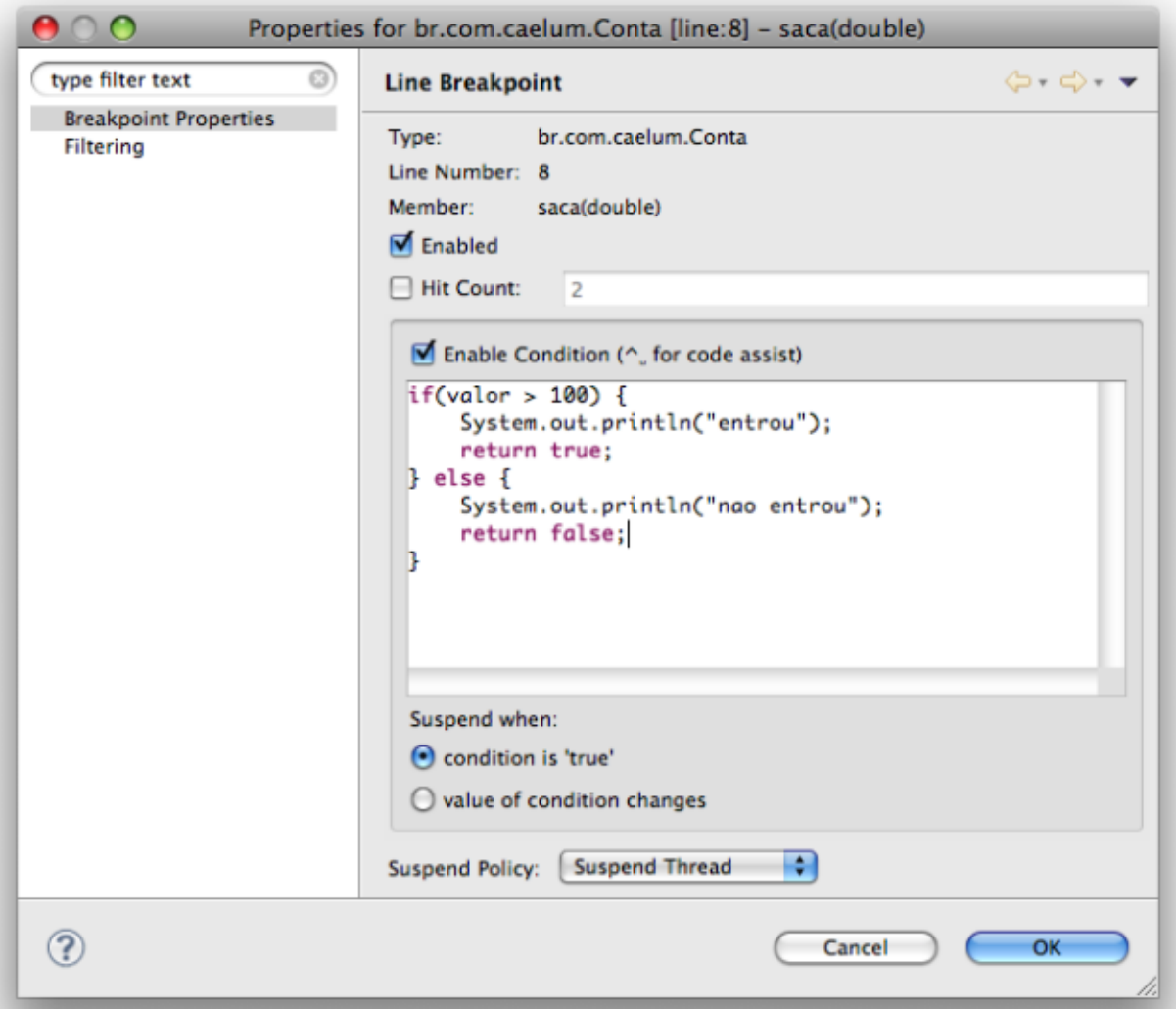
Você também pode usar os botões que estão presentes na aba `Debug`.

22.4 - Debug avançado

Depois que colocamos um breakpoint em algum ponto do nosso código, podemos colocar algumas propriedades nele, por exemplo, usar alguma condição para restringir quando o breakpoint será ativado em tempo de execução. Podemos restringir na propriedade **Hit Count** que o breakpoint só será ativado quando a linha em que ele encontra-se for executada 'X' vezes.



Como na imagem acima o breakpoint só será ativado quando a linha de código em que ele se encontra for executada '2' vezes. Podemos também colocar alguma expressão condicional, um `if`, por exemplo.



O breakpoint, neste caso, somente será ativado quando o argumento `valor` que foi passado ao método `saca` for maior que 100. O importante aqui é notarmos que devemos retornar **sempre** um valor booleano, se não o fizermos, teremos um erro em tempo de execução. Essa propriedade é válida quando queremos colocar aqueles famosos `System.out.println("entrou no if tal")` para efeito de log, podemos fazer isso colocando o log dentro da expressão condicional nas propriedades do breakpoint.

O display é uma das partes mais interessantes do debug do eclipse, ele provê uma maneira de executarmos qualquer código que quisermos quando estamos em debugging. Criar uma classe, instanciar objetos dessa classe, utilizar if's, for's, while's, todos os recursos do Java, além de poder utilizar as variáveis, métodos, constantes da classe que estamos debugando.

Um exemplo clássico é quando estamos em debugging e queremos saber o retorno de algum método do qual não temos acesso, o que faríamos antes seria colocar um amontoado de `System.out.println`, poluindo extremamente nosso

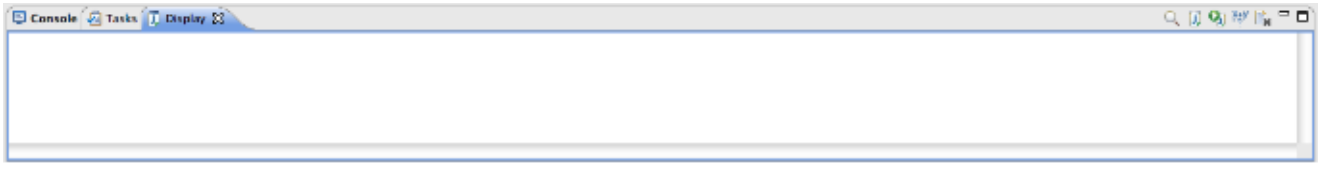
código. No display o que fazemos é efetuar a chamada desse código e automaticamente os resultados são exibidos.

Para vermos um efeito real disso, vamos alterar um pouco o comportamento da classe Conta, de modo que agora o saldo para saque tenha que ser o saldo real mais o valor do limite. Nosso código fica assim:

```
public class Conta {

  private double saldoReal;
  private double limite;

  public Conta(double limite) {
    this.limite = limite;
  }

  public boolean saca(double valor) {
    if (!isSaldoSuficiente(valor)) {
      return false;
    } else {
      this.saldoReal = this.saldoReal - valor;
      return true;
    }
  }

  private boolean isSaldoSuficiente(double valor) {
    return (this.saldoReal + this.limite) > valor;
  }
}
```

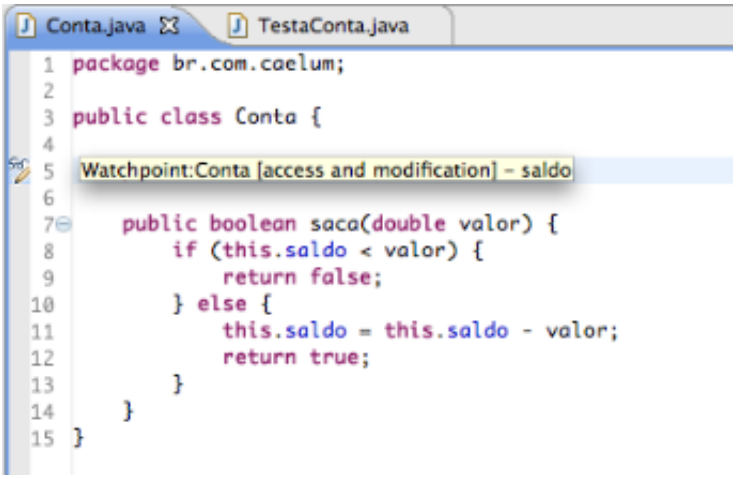
Repare que o `if` que verifica se o saldo é suficiente para efetuarmos o saque chama um método `isSaldoSuficiente`, o que pode ser um problema quando estamos debugando, afinal a condição do `if` é um método. Se utilizarmos o display podemos fazer a chamada do método `isSaldoSuficiente`, ver seu resultado e o melhor, não afetamos o debug, apenas queremos ver o resultado do método, por exemplo.

Para exibirmos a aba **Display** é bem simples. Tecle **Ctrl + 3**, digite **Display** e a aba será exibida. Quando rodarmos nosso código em modo debug, podemos ir no display, digitarmos uma chamada para o método `isSaldoSuficiente`, executamos esse código que foi digitado selecionando-o dentro do display e teclando **Ctrl + Shift + D** e o resultado será impresso, assim como na imagem abaixo:

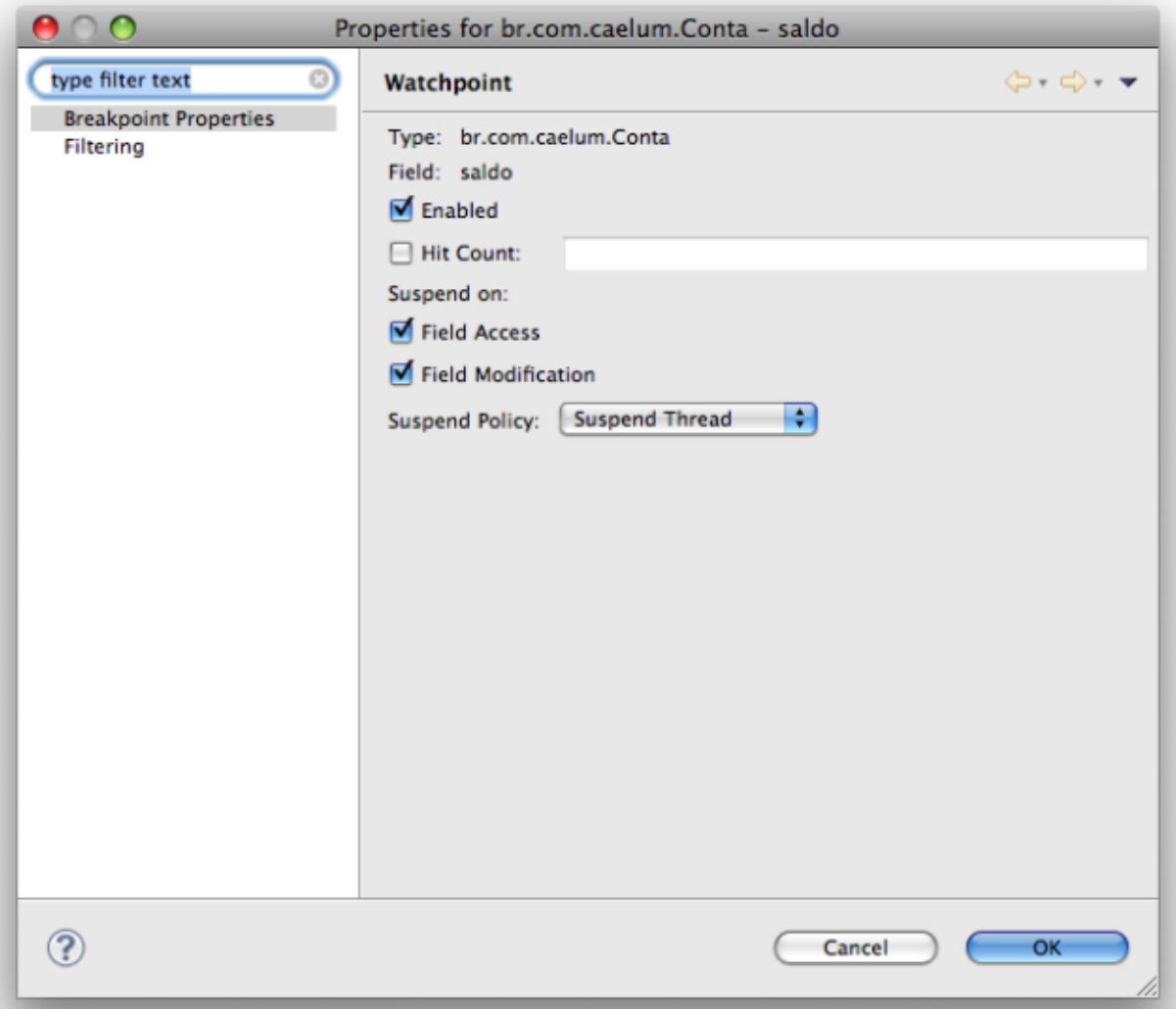


Muitas vezes queremos "seguir" alguma variável de instância, ou seja, qualquer

chamada para essa variável (leitura ou escrita) queremos ser notificados disso. Podemos usar o watchpoint, que fará nosso programa entrar em modo debug, quando qualquer alteração na variável que estamos seguindo ocorrer, o programa entrará em debug exatamente na linha que fez a alteração. Para colocarmos um watchpoint, basta dar um duplo clique no atributo de instância que deseja colocá-lo.



É possível alterar esse comportamento padrão, e definir se você quer que o watchpoint seja ativado para leitura ou somente para escrita.

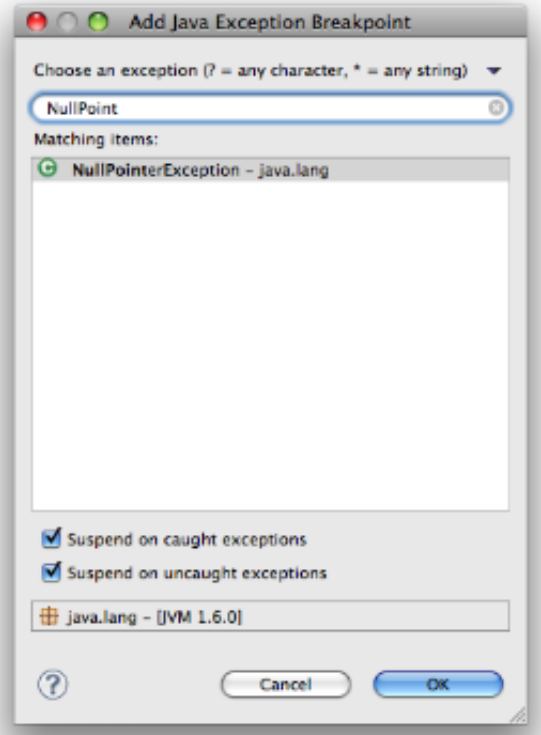


A idéia desse tipo de breakpoint é fazer nosso programa entrar em debug quando alguma exceção específica ocorrer. Quando definirmos essa exceção no **Exception Breakpoint** e a mesma ocorrer, automaticamente nosso programa entra em debug na linha que gerou aquela exceção. Por exemplo, vamos alterar o codigo da classe `TestaConta` para que a mesma tenha uma `NullPointerException`:

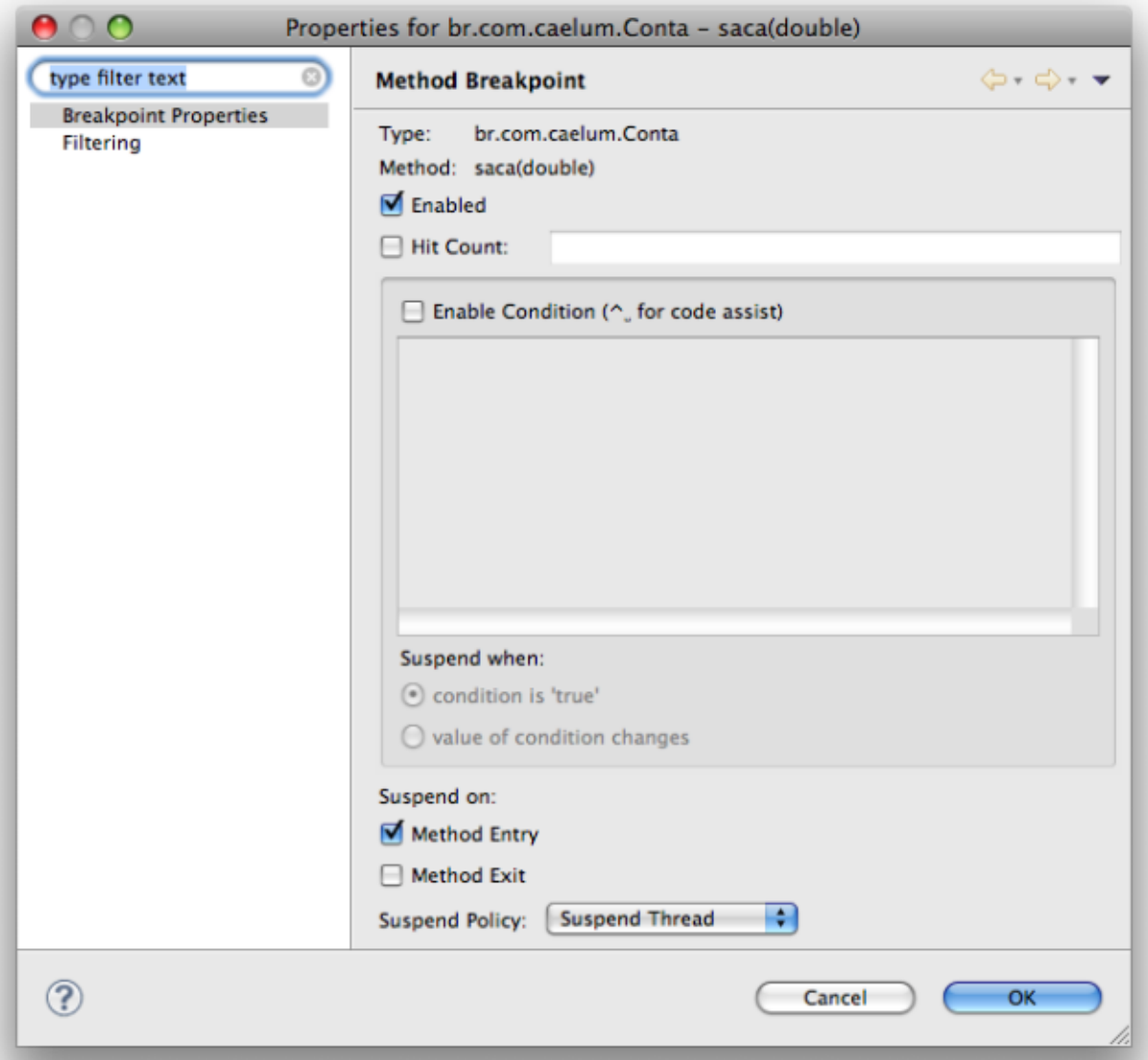
```
public class TestaConta {
  public static void main(String[] args) {
    Conta conta = null;
    conta.saca(10);
  }
}
```

Quando rodarmos o código acima, teremos uma `NullPointerException`. Pode ser útil nesses casos debugar e saber onde a exceção está ocorrendo de fato, em qual linha mais especificamente. Para fazermos isso podemos criar um Exception Breakpoint, que debugará códigos que eventualmente lancem uma `NullPointerException`, por exemplo. Basta abrirmos a aba **Breakpoints** e clicarmos no ícone abaixo:

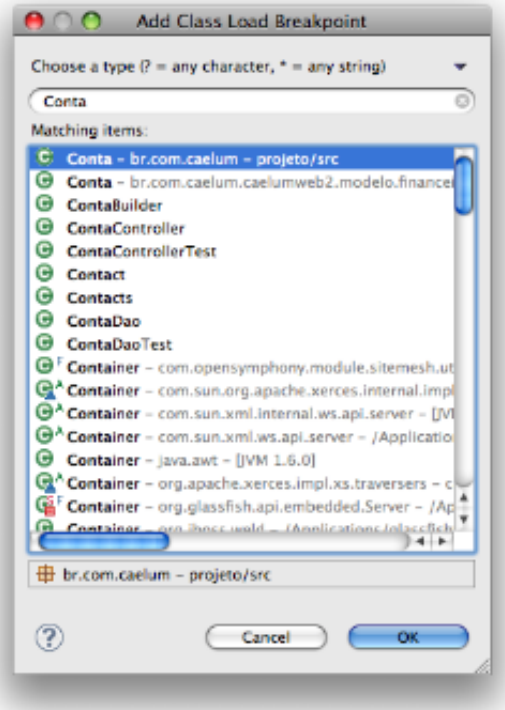
Será aberta uma janela onde podemos buscar por uma exceção específica.



Podemos definir um breakpoint que é ativado ou antes ou depois que o método é chamado. Para definirmos ele, basta estar em qualquer parte do método que desejamos debugar, clicar no menu **Run -> Toogle Method Breakpoint**. Podemos editar as propriedades desse breakpoint dizendo se queremos que ele seja ativado antes(default) ou depois da execução do método. Basta acessar as propriedades do method breakpoint e alterá-las.



É util quando desejamos que um breakpoint seja ativado quando uma classe específica for carregada pela primeira vez, chamamos esse breakpoint de **Class Breakpoint**. Basta clicarmos no menu **Run -> Add Class Load Breakpoint**, uma janela será aberta e basta digitarmos o nome da classe e adicionarmos:



22.5 - Profiling

Um dos principais hábitos que nós desenvolvedores devemos evitar é a questão da otimização prematura, ou seja, quando desenvolvemos uma aplicação para um cliente, devemos nos preocupar em **atender o requisitos funcionais de maneira mais rápida e mais simples possível**. O passo seguinte é refatorar seu código para que ele seja melhorado e para que no futuro possa se adaptar as possíveis mudanças.

A regra é: "Deixe os problemas do futuro, para serem resolvidos no futuro".

Uma das ferramentas que nos auxiliam na questão de não otimizar nosso código prematuramente, são as ferramentas de profiling, que tornam aparentes, por exemplo, os problemas de memória e cpu, que podem fazer com que otimizemos nosso código. Atualmente devido as técnicas que utilizamos para entregar algo de valor para o cliente, focamos principalmente na qualidade, aspectos funcionais, testes, etc. Porém, muitos problemas que não fazem parte dos requisitos funcionais podem acontecer apenas quando a aplicação está em produção, neste ponto as ferramentas de profiling também nos ajudam.

português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

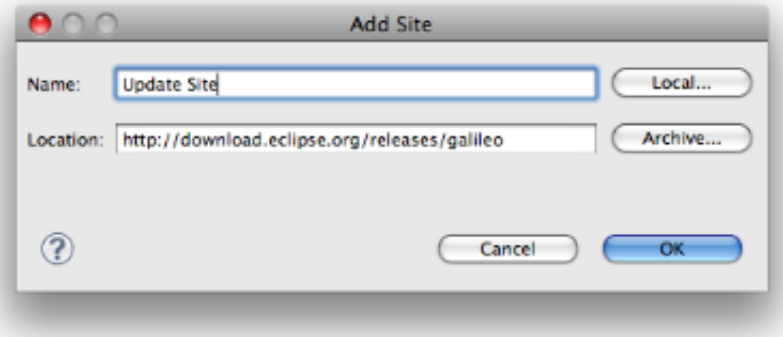
22.6 - Profiling no Eclipse TPTP

Juntamente com o Eclipse temos a opção de instalar e utilizar uma ferramenta de profiling conhecida como Eclipse TPTP (Eclipse Test & Performance Tools Platform), que nos fornece opções para isolar e identificar problemas de performance, tais como: memória (memory leak), recursos e processamento. O TPTP nos permite analisar de simples aplicações java até aplicações que rodam em múltiplas máquinas e em múltiplas plataformas.

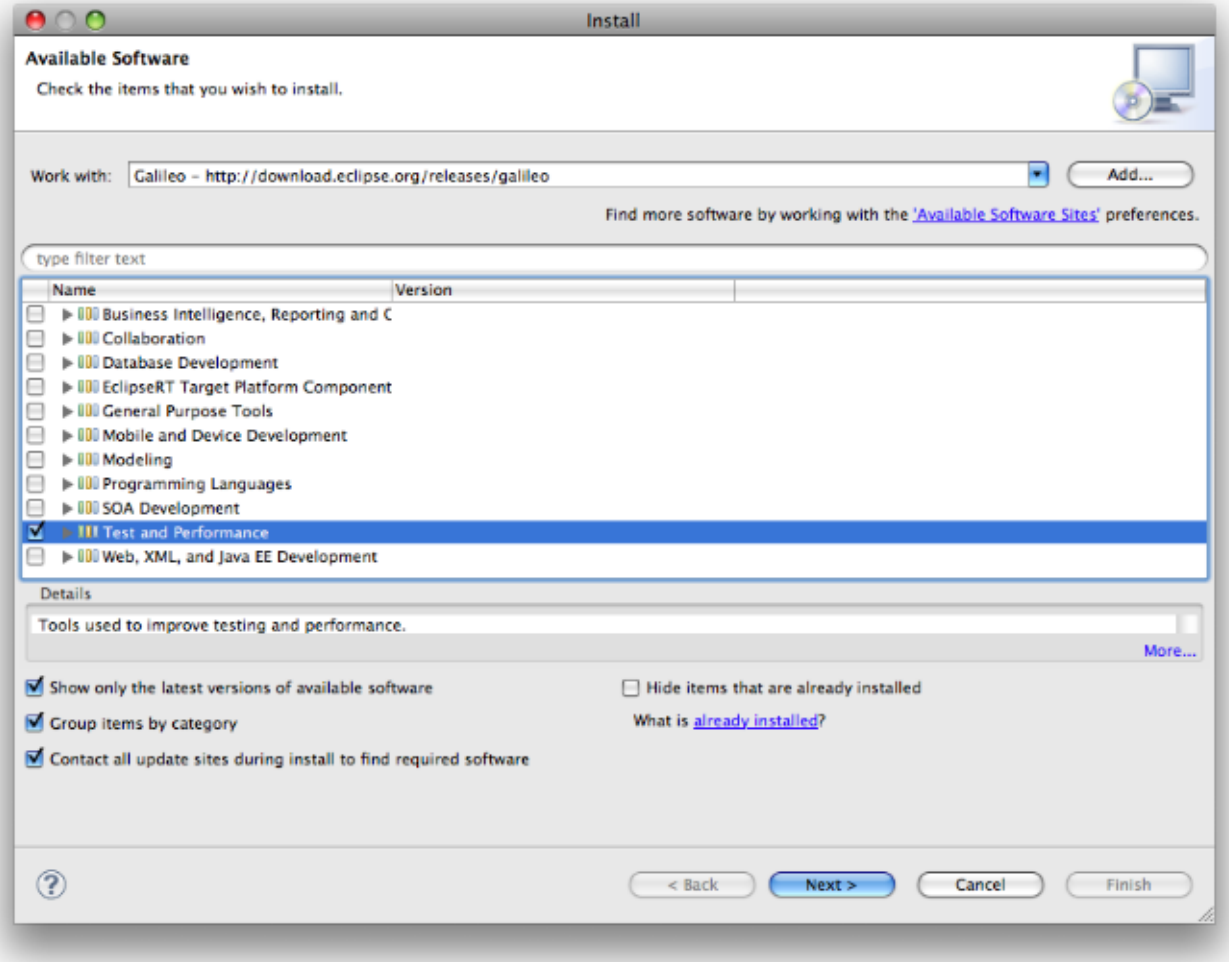
Alternativas ao TPTP

Existem algumas alternativas ao TPTP, os mais conhecidos são Netbeans Profiler (http://profiler.netbeans.org/) que é gratuito, e o JProfiler (http://www.ej-technologies.com/products/jprofiler/overview.html) que é pago.

O TPTP não vem por padrão junto com o Eclipse. Portanto, para utilizarmos é necessário a instalação do mesmo. Podemos fazer o processo de instalação de duas maneiras. A primeira e mais fácil é utilizando o Update Site do Eclipse que resolve as possíveis dependências e nos possibilita escolher quais features queremos instalar. Para instalar o TPTP através desse recurso, basta ir no menu: *Help -> Install New Software*, uma janela será aberta, basta clicar em Add... e preenchê-la conforme a imagem a seguir:



Basta adicionar as ferramentas do TPTP em nosso eclipse, para isto, selecione o repositório que acabamos de adicionar e a versão do TPTP que queremos instalar, neste caso, a versão 4.6.2.



Instalando pelo Zip

Você tem a opção de instalar o TPTP baixando o zip do projeto e colocando manualmente no diretório de instalação do seu eclipse. Mais informações no link: http://www.eclipse.org/tptp/home/downloads/4.6.0/documents/installguid

e/InstallGuide46.html

Um problema que pode acontecer em aplicações e que muitas pessoas não conhecem a fundo, é a questão do pool de Strings que pode eventualmente ficar muito grande. Este problema pode ser causado porque objetos do tipo String são imutáveis, sendo assim, se fizermos concatenações de Strings muitas vezes, cada uma dessas concatenações produzirá uma nova String, que automaticamente será colocada no pool da JVM.

A alternativa neste caso, seria trabalhar com objetos do tipo StringBuilder ou StringBuffer que funcionam como Strings, mas que não produzem Strings novas em caso de uma concatenação. Mas como medir o tamanho do nosso pool de String?

O TPTP possui uma aba de estatísticas que nos mostra o tempo que um método levou para ser executado, quanto processamento esse método gastou, quanto de memória foi gasto com cada método. Vamos analisar algumas dessas estatísticas criando um código que concatene várias Strings, de maneira que sobrecarregue o pool, gere bastante processamento e consumo de memória.

```
public class Teste {

  public static void main(String[] args) {
    for (int i = 0; i < 1000000; i++) {
      String x = "a" + i;
      System.out.println(x);
    }
  }

}
```

Para analisarmos o resultado do código, vamos rodar o código do `main` através do menu *Run -> Profile As -> Java Application.*

Versões

Infelizmente o TPTP funciona somente no Windows. Versões para MacOS e Linux são prometidas, mas até hoje estão em desenvolvimento. Uma alternativa paga para esses outros sistemas operacionais é o JProfiler.

CAPÍTULO ANTERIOR:

Apêndice – Instalação do Java

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter