Com certeza, todo desenvolvedor de software já escreveu um trecho de código que não funcionava. E pior, muitas vezes só descobrimos que o código não funciona quando nosso cliente nos reporta o bug. Nesse momento, perdemos a confiança no nosso código (já que o número de bugs é alto) e o cliente perde a confiança na equipe de desenvolvimento (já que ela não entrega código de qualidade). Mas será que isso é difícil de acontecer?

Para exemplificar isso, imagine que hoje trabalhamos em um sistema de leilão. Nesse sistema, um determinado trecho de código é responsável por devolver o maior lance de um leilão. Veja a implementação deste código:

```
class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;

    public void avalia(Leilao leilao) {

        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos) {
                maiorDeTodos = lance.getValor();
            }
        }
    }

    public double getMaiorLance() {
        return maiorDeTodos;
    }
}
```

```
class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));
        leilao.propoe(new Lance(maria,250.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        System.out.println(leiloeiro.getMaiorLance()); // imprime 400.0
    }
}
```

Esse código funciona. Ao receber um leilão, ele varre a lista buscando o maior valor. Veja que a variável `maiorDeTodos` é inicializada com o menor valor que cabe em um `double`. Isso faz sentido, já que queremos que, na primeira vez que o `for` seja executado, ele caia no `if` e substitua o menor valor do double pelo primeiro valor da lista de lances.

A próxima funcionalidade a ser implementada é a busca pelo menor lance de todos. Essa regra de negócio faz sentido estar também na classe `Avaliador`. Basta acrescentarmos mais uma condição, desta vez para calcular o menor valor:

```
class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;
    private double menorDeTodos = Double.POSITIVE_INFINITY;

    public void avalia(Leilao leilao) {

        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos) {
                maiorDeTodos = lance.getValor();
            }
            else if(lance.getValor() < menorDeTodos) {
                menorDeTodos = lance.getValor();
            }
        }
    }

    public double getMaiorLance() { return maiorDeTodos; }
    public double getMenorLance() { return menorDeTodos; }
}
```

```
class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));
        leilao.propoe(new Lance(maria,250.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        System.out.println(leiloeiro.getMaiorLance()); // imprime 400.0
        System.out.println(leiloeiro.getMenorLance()); // imprime 250.0
    }
}
```

Tudo parece estar funcionando. Apareceram na tela o menor e maior valor corretos. Vamos colocar o sistema em produção, afinal está testado!

Mas será que o código está realmente correto? Veja agora um outro teste, muito parecido com o anterior:

```
class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        System.out.println(leiloeiro.getMaiorLance()); // imprime 400.0
        System.out.println(leiloeiro.getMenorLance()); // INFINITY
    }
}
```

E veja que, para um cenário um pouco diferente, nosso código não funciona! A grande pergunta é: no mundo real, será que teríamos descoberto esse bug facilmente, ou esperaríamos nosso cliente nos ligar bravo porque a funcionalidade não funciona? Infelizmente, bugs em software são uma coisa mais comum do que deveria ser. Bugs nos fazem perder a confiança do cliente, e nos custam muito dinheiro. Afinal, precisamos corrigir o bug e recuperar o tempo perdido do cliente, que ficou parado, enquanto o sistema não funcionava.

Por que nossos sistemas apresentam tantos bugs assim? Um dos motivos para isso é a falta de testes, ou seja, testamos muito pouco! Equipes de software geralmente não gostam de fazer (ou não fazem) os devidos testes. As razões para isso são geralmente a demora e o alto custo para testar o software como um todo. E faz todo o sentido: pedir para um ser humano testar todo o sistema é impossível: ele vai levar muito tempo para isso!

E como resolver esse problema? Fazendo a máquina testar! Escrevendo um programa que teste nosso programa de forma automática! Uma máquina, com certeza, executaria o teste muito mais rápido do que uma pessoa! Ela também não ficaria cansada ou cometeria erros!

Um teste automatizado é muito parecido com um teste manual. Imagine que você está testando manualmente uma funcionalidade de cadastro de produtos em uma aplicação web. Você, com certeza, executará três passos diferentes. Em primeiro lugar, você pensaria em um cenário para testar. Por exemplo, "vamos ver o que acontece com o cadastro de funcionários quando eu não preencho o campo de CPF". Após montar o cenário, você executará a ação que quer testar. Por exemplo, clicar no botão "Cadastrar". Por fim, você olharia para a tela e verificaria se o sistema se comportou da maneira que você esperava. Nesse nosso caso, por exemplo, esperaríamos uma mensagem de erro como "CPF inválido".

Um teste automatizado é muito parecido. Você sempre executa estes três passos: monta o cenário, executa a ação e valida a saída. Mas, acredite ou não, já escrevemos algo muito parecido com um teste automatizado nesta aula. Lembra da nossa classe `Teste`? Veja que ela se parece com um teste, afinal ela monta um cenário e ela executa uma ação. Veja o código abaixo:

```
class Teste {

    public static void main(String[] args) {
        // cenario: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        // executando a acao
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // exibindo a saida
        System.out.println(leiloeiro.getMaiorLance()); // imprime 400.0
        System.out.println(leiloeiro.getMenorLance()); // imprime INFINIT
    }
}
```

E veja que ele é automatizado! Afinal, é a máquina que monta o cenário (nós escrevemos o código, sim, mas na hora de executar, é a máquina que executa!), e é ela que executa a ação. Não gastamos tempo nenhum para executar esse teste. O problema é que ele ainda não é inteiro automatizado. A parte final do teste (a validação) ainda é manual: o programador precisa ver o que o programa imprimiu na tela e ver se o resultado bate com o esperado.

Para melhorar isso, precisamos fazer a própria máquina verificar o resultado. Para isso, ela precisa saber qual a saída esperada. Ou seja, a máquina deve saber que o maior esperado, para esse caso, é 400, e que o menor esperado é 250. Vamos colocá-la em uma variável e então pedir pro programa verificar se a saída é correta:

```
class TesteDoAvaliador {

    public static void main(String[] args) {
        // cenario: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        // executando a acao
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // comparando a saida com o esperado
        double maiorEsperado = 400;
        double menorEsperado = 250;

        System.out.println(maiorEsperado == leiloeiro.getMaiorLance());
        System.out.println(menorEsperado == leiloeiro.getMenorLance());
    }
}
```

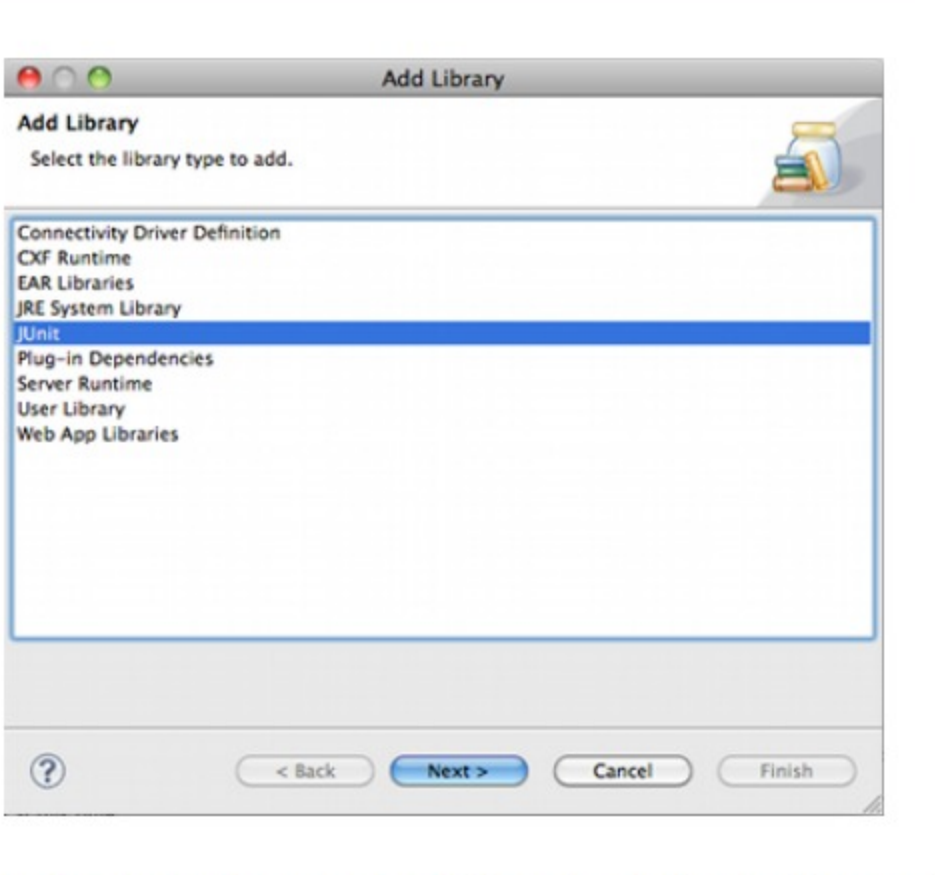
Ao rodar esse programa, a saída já é um pouco melhor:

```
true
false
```

Veja que agora ela sabe o resultado esperado e verifica se a saída bate com ele, imprimindo `true` se o resultado bateu e `false` caso contrário. Estamos chegando perto. O desenvolvedor ainda precisa olhar todos os trues e falses e ver se algum deu errado. Imagina quando tivermos 1000 testes iguais a esses? Será bem complicado!

Precisávamos de uma maneira mais simples e elegante de verificar o resultado de nosso teste. Melhor ainda se soubéssemos exatamente quais testes falharam e por quê. Para resolver esse problema, utilizaremos o JUnit, o framework de testes de unidade mais popular do mundo Java. O JUnit é uma ferramenta bem simples. Tudo que ele faz é te ajudar na automatização da última parte de um teste, a validação, e a exibir os resultados de uma maneira bem formatada e simples de ser lida.

Para facilitar a interpretação do resultado dos testes, o JUnit pinta uma barra de verde, quando tudo deu certo, ou de vermelho, quando algum teste falhou. Além disso, ele te mostra exatamente quais testes falharam e qual foi a saída incorreta produzida pelo método. O JUnit é tão popular que já vem com o Eclipse. Para adicionar a biblioteca ao seu projeto, basta clicar com o botão direito no projeto e selecionar a opção `Build Path -> Add Libraries -> JUnit -> Versão 4`.



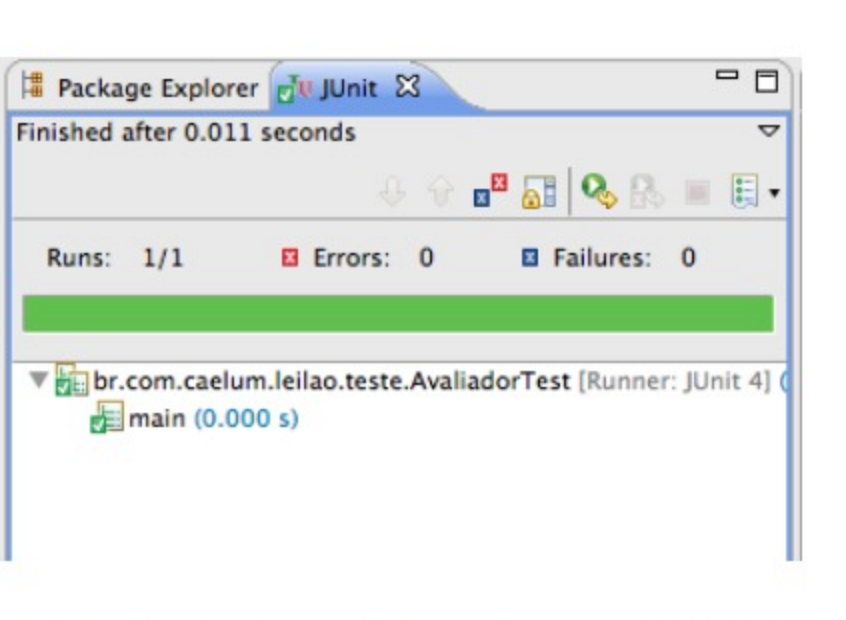
Nosso código acima está muito perto de ser entendido pelo JUnit. Precisamos fazer apenas algumas mudanças: 1) um método de teste deve sempre ser público, de instância (isto é, não pode ser `static`) e não receber nenhum parâmetro; 2) deve ser anotado com `@Test`. Vamos fazer estas alterações:

```
class AvaliadorTest {

    @Test
    public void main() {
        // cenario: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        // executando a acao
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // comparando a saida com o esperado
        double maiorEsperado = 400;
        double menorEsperado = 250;

        System.out.println(maiorEsperado == leiloeiro.getMaiorLance());
        System.out.println(menorEsperado == leiloeiro.getMenorLance());
    }
}
```

Repare algumas coisas no código acima. Veja que mudamos o nome da classe: agora ela se chama `AvaliadorTest`. É convenção que o nome da classe geralmente seja NomeDaClasseSobTesteTest, ou seja, o nome da classe que estamos testando (no caso, `Avaliador`) mais o sufixo Test.

Seguindo a ideia da convenção do nome da classe, vamos querer colocar mais métodos para testar outros cenários envolvendo essa classe. Mas veja o nome do método do teste: continua `main`. Será que esse é um bom nome? Que nome daremos para os outros testes que virão?

Quando rodamos os testes pelo JUnit, ele nos mostra o nome do método que ele executou e um ícone indicando se aquele teste passou ou não.



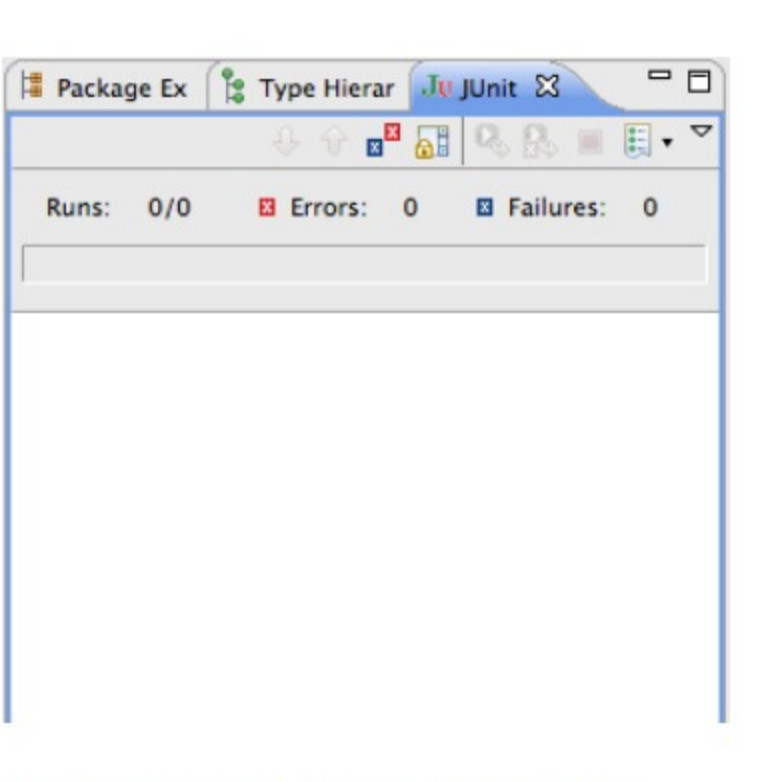
Olhando para a figura acima, dá para saber o que estamos testando? Ou então que parte do sistema está quebrada? Os nomes dos testes não nos dão nenhuma dica sobre o que está sendo testado. Precisamos ler o código de cada teste para descobrir. O ideal seria que os nomes dos testes já nos dessem uma boa noção do que estamos testando em cada método. Assim, conseguimos descobrir mais facilmente o que está quebrado no nosso sistema. Vamos, então, renomear o método:

```
class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
        // cenario: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        // executando a acao
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // comparando a saida com o esperado
        double maiorEsperado = 400;
        double menorEsperado = 250;

        System.out.println(maiorEsperado == leiloeiro.getMaiorLance());
        System.out.println(menorEsperado == leiloeiro.getMenorLance());
    }
}
```

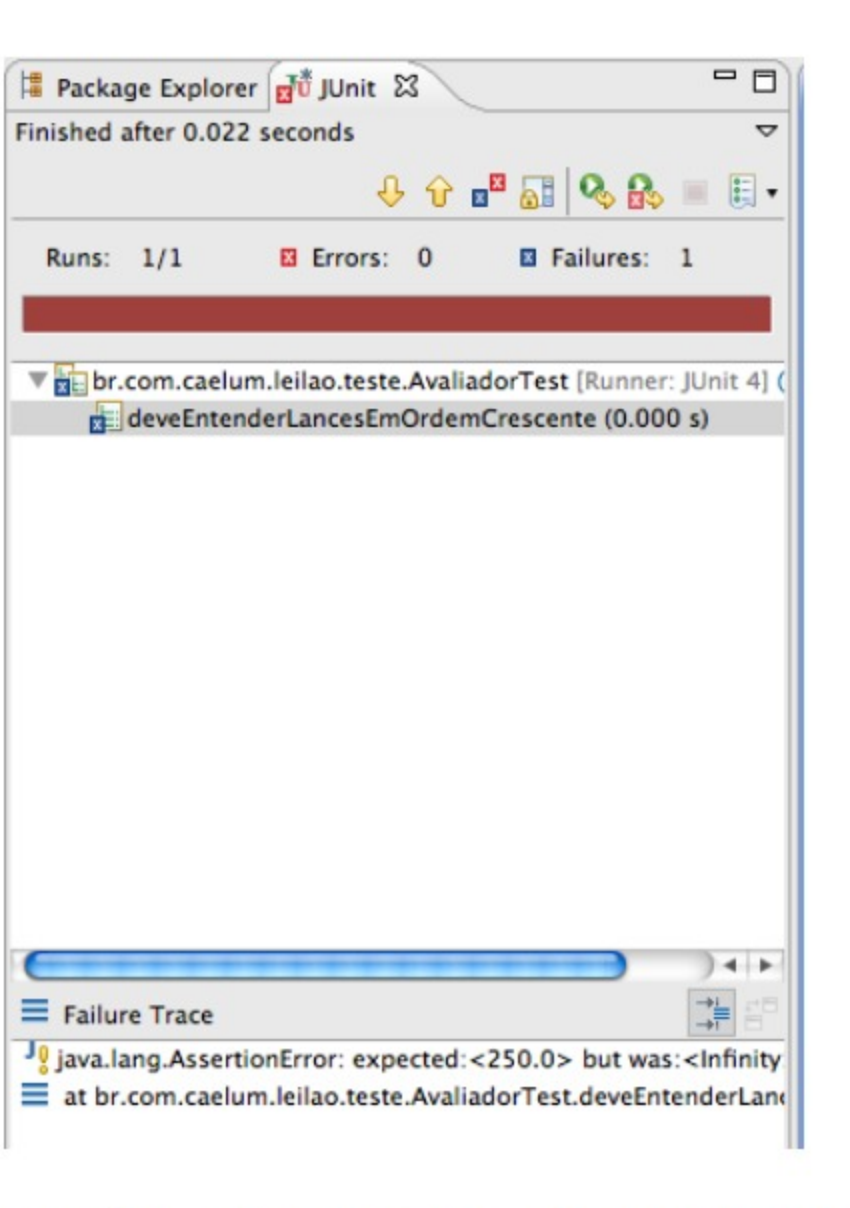
A última coisa que precisamos mudar no código para que ele seja entendido pelo JUnit são os `System.out.println()`. Não podemos imprimir na tela, pois assim nós é que seríamos responsáveis por validar a saída. Quem deve fazer isso agora é o JUnit! Para isso, utilizaremos a instrução `Assert.assertEquals()`. Esse é o método utilizado quando queremos que o resultado gerado seja igual a saída esperada. Em código:

```
import org.junit.Assert;

class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
        // cenario: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        // executando a acao
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // comparando a saida com o esperado
        double maiorEsperado = 400;
        double menorEsperado = 250;

        Assert.assertEquals(maiorEsperado, leiloeiro.getMaiorLance(), 0.0
        Assert.assertEquals(menorEsperado, leiloeiro.getMenorLance(), 0.0
    }
}
```

Repare que importamos a classe `Assert` e utilizamos o método duas vezes: para o maior e para o menor lance. Vamos agora executar o teste! Para isso, basta clicar com o botão direito no código da classe de teste e selecionar `Run as -> JUnit Test`. Veja que a view do JUnit se abrirá no Eclipse com o resultado do teste.



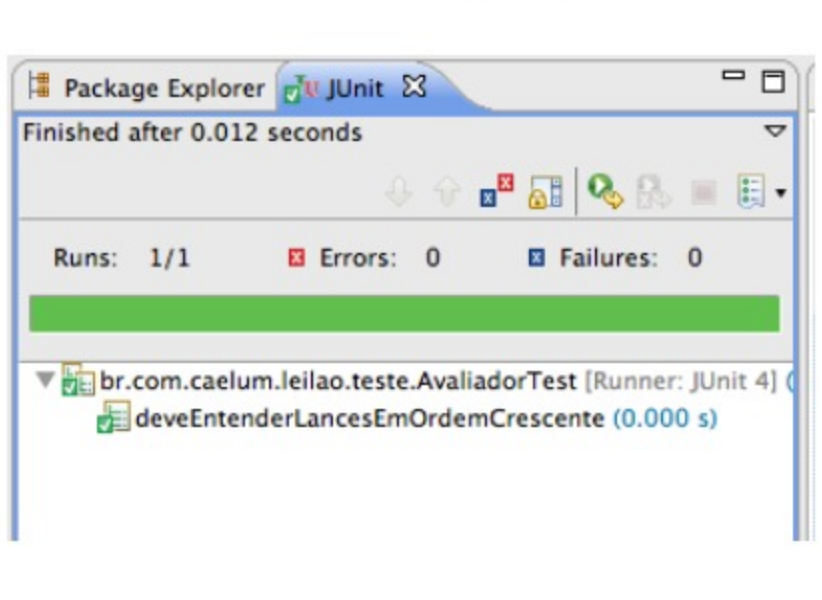
Nosso teste não passa. Veja a mensagem de erro dada pelo JUnit:



Vamos corrigir o bug. No nosso código, temos um `else` sobrando! Ele faz toda a diferença. Vamos corrigir o código:

```
class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;
    private double menorDeTodos = Double.POSITIVE_INFINITY;

    public void avalia(Leilao leilao) {

        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos) {
                maiorDeTodos = lance.getValor();
            }
            if(lance.getValor() < menorDeTodos) {
                menorDeTodos = lance.getValor();
            }
        }
    }

    public double getMaiorLance() { return maiorDeTodos; }
    public double getMenorLance() { return menorDeTodos; }
}
```

Vamos rodar o teste novamente. Agora ele está verde: passou!



Veja o tempo que o teste levou para ser executado: alguns milissegundos. Isso significa que podemos rodar esse teste O TEMPO TODO, que é rápido e não custa nada. É a máquina que roda! Agora imagine um sistema com 5000 testes como esses. Ao clicar em um botão, o JUnit, em alguns segundos, executará 5000 testes! Quanto tempo levaríamos para executar o mesmo teste de maneira manual?