# Mais ajuda com Java - Fazendo o programa compilar e funcionar

Durantes as últimas três atividade de Java 2 grandes dificuldades apareceram relacionadas a Java.

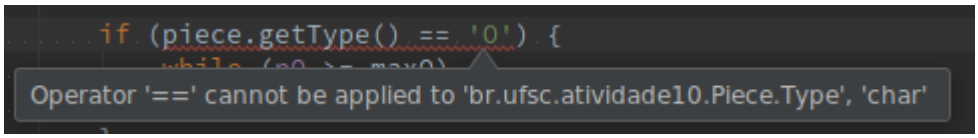
1. Escrever programas que compilam
2. Escrever programas que funcionam

O Conteúdo abaixo está divido em duas partes, cada uma apresenta uma ferramenta que ajuda a resolver a dificuldade (disclaimer: a JetBrains não me pagou).

# Sintaxe / Erros do compilador

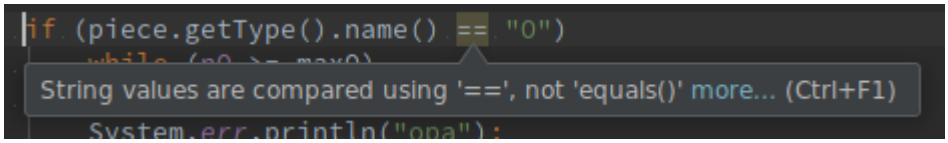
Não vou conseguir ensinar tudo que se supunha que vocês soubessem de programação após um ano de curso em um email. O que posso fazer por vocês é dar duas dicas. Se o problema de vocês é com sintaxe (o programa não compila) recomendo fortemente que instalem o eclipse ou o IntelliJ IDEA. Essas IDEs não apenas consomem toda a RAM do sistema, elas esfregam nossos erros na nossa cara. Vocês podem não gostar do sublinahdo vermelho e das esclamações pipocando pela tela. Mas a triste verdade é que se vocês não vêm essas coisas, não quer dizer que esteja certo.

O primeiro erro que vou mostrar parece surgir de uma tentativa de programar em Java como se estivesse em JavaScript (que não tem nada a ver com Java).




O método Piece.getType() retorna um Piece.Type (percebam que o case das letras importa!). A expressão 'O' tem tipo char. O compilador não sabe como aplicar o operador == a valores desses tipos, pois não há como comparar essas coisas. O que deve ser feito é comprar com uma valor do próprio enum: `piece.getType() == Piece.Type.O`

Uma solução alternativa errada (pegadinha pra novatos em Java) seria fazer isso:




Aqui é importante lembrar da distinção entre primitivos (int, char, double, float, long short, char e byte) e Objetos (instâncias de classes ou enums). Em Java, == e != funcionam para primitivos e para objetos, mas com singnificados completamente diferentes

- Entre primitivos, == e != comparam o **valor**: int x = 2; x == 2
- Entre objetos, == e != comparam a **referência**. Por debaixo dos panos, toda variável em Java que não seja um primitivo é um ponteiro. Na expressão problemática da última imagem, temos dois ponteiros: o retornado por piece.getType().name() e o retornado por "O". Embora os objetos apontados sejam objetos iguais, cada lado do == aponta para um objeto diferente, e por isso, a comparação irá falhar sempre!

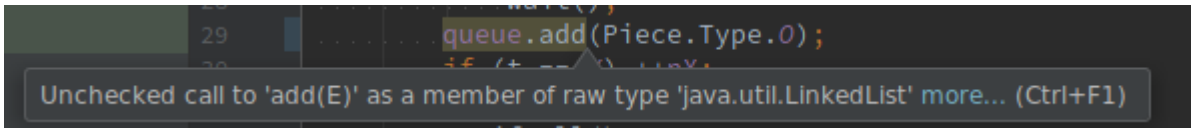
Nesse caso a solução correta seria:

```
if (piece.getType() == Piece.Type.O) {
    //...
}
```

Mas se por algum motivo eu quisesse usar comparação de strings, deveria usar o método equals:

```
if (piece.getType().name().equals("O")) {
    //...
}
```

O próximo erro é um pouco mais sofisticado, ele está disfarçado atrás de um warning:



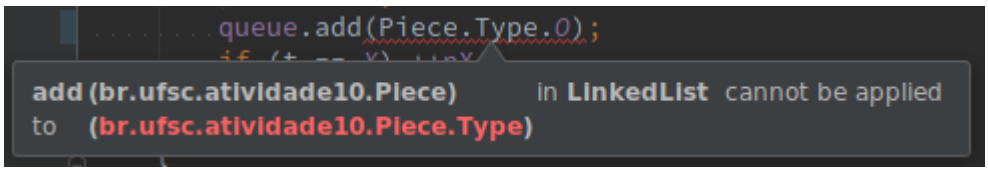

Esse warning está me dizendo que o compilador está passando o objeto piece (que é do tipo Piece) para o método add() sem checar se o tipo de piece é realmente comaptível com o tipo do argumento recebido por add(). Isso não é um erro, mas apenas um warning devido a forma como generics (e.g., List<String> -- uma lista **de** strings) são implementados na linguagem Java. O compilador suspeita que estamos fazendo uma chamada inválida, mas ele não consegue ter provas o suficiente pra disparar um erro. Nesse caso o motivo do erro está na própria mensagem. Observem que ela diz java.util.LinkedList e não java.util.LinkedList<Piece>. Seguindo a definição da variável queue, encontrei isso:

```
private LinkedList queue = new LinkedList();
```

que deveria ser trocado por isso:

```
private LinkedList<Piece> queue = new LinkedList<>();
```

Corrigido esse erro na definição da variável o warning anterior se transforma e um erro. E isso é bom:




O compilador está dizendo que eu tentei colocar uma instância do enum `Piece.Type` em um lugar onde só é aceito um `Piece`. Nesse caso a solução é passar a variável piece, como na solução da atividade.

# Funcionamento do programa/Debug

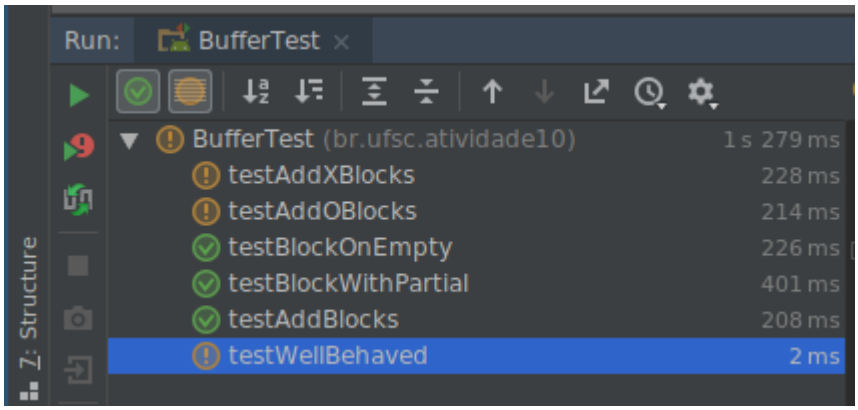
Observo o comportamento de vocês na sala de aula e percebo que quando o programa de vocês não funciona vocês entram em um ciclo de encarar o código com a mão no queixo por vários segundos  e então scrollar os slides.É importante refletir sobre as escolhas, mas se vocês estão com dificuldade de entender a linguagem, talvez seja melhor ver o programa em funcionamento.

As IDEs Java tem uma ferramenta fantástica, chamda debugger que vale cada megabyte de RAM ocupado. Se você nunca usou um debugger em Java, o dia de hoje ficará marcado como o fim de uma era de escuridão e sofrimento.

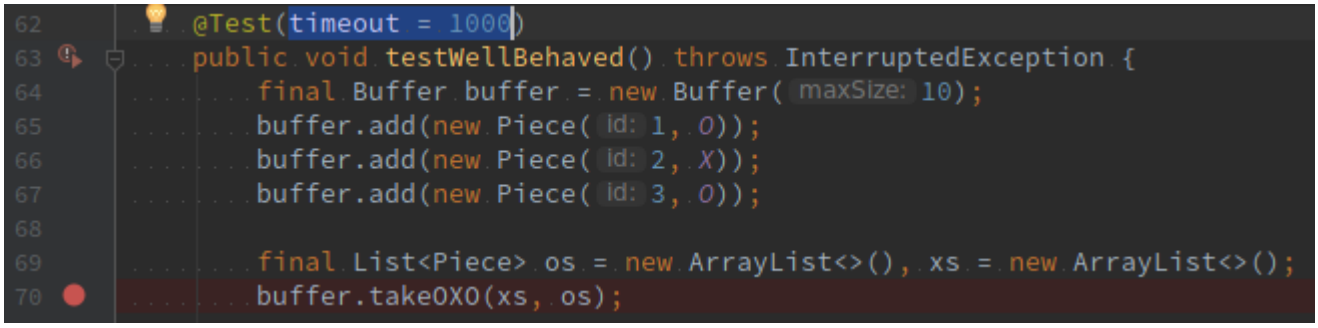
Para entender como usar o debugger, pegue a solução da atividade de monitores e estrague o método takeOXO na linha 41 trocando um || por um &&:

```
Iterator<Piece> it = queue.iterator();
while (it.hasNext() && xList.size() < 1 && oList.size() < 2) {
    Piece piece = it.next();
    if (piece.getType() == X && xList.size() < 1) {
        xList.add(piece);
        it.remove();
        --nX;
    } else if (piece.getType() == O && oList.size() < 2) {
        oList.add(piece);
        it.remove();
        --nO;
    }
}
notifyAll();
```

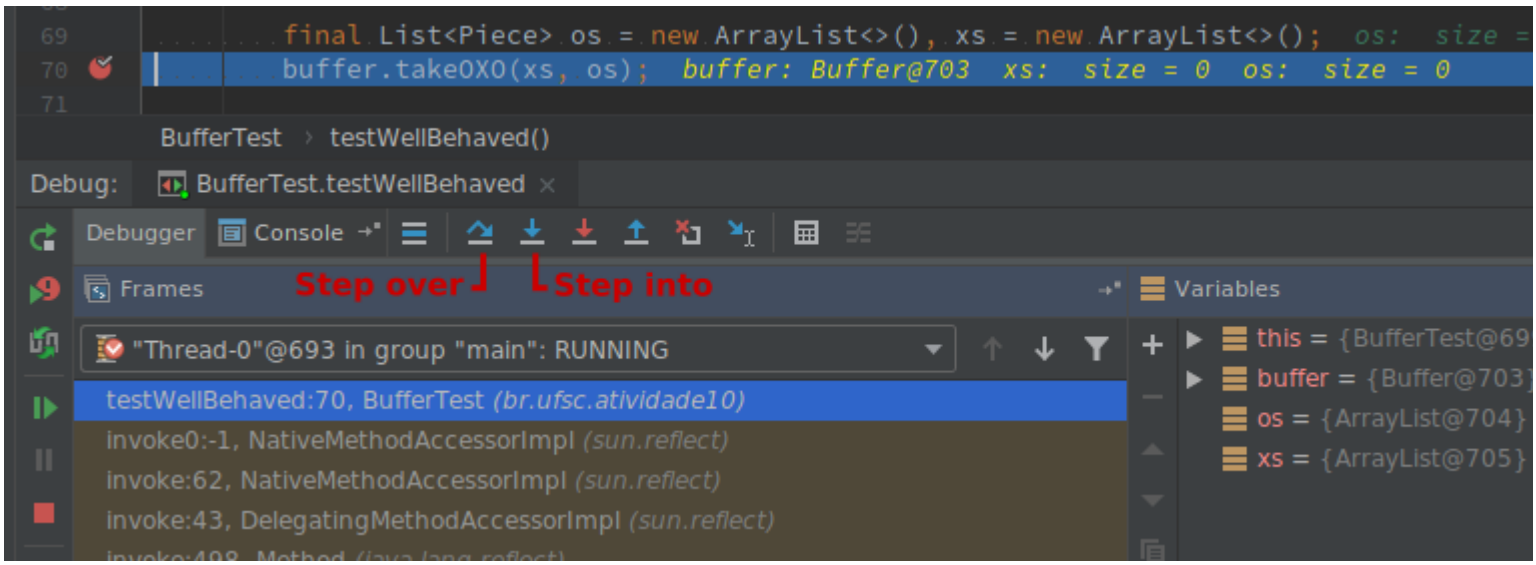
Ao rodar os testes, você constatará que realmente o código não está se comportando bem:




Vá até o teste que falhou e insira um *breakpoint* clicando na margem esquerda da área de código (No IntelliJ é um clique, no eclipse são dois). Como esse teste tem um timeout, é importante remover ele (apague a parte selecionada em azul) :



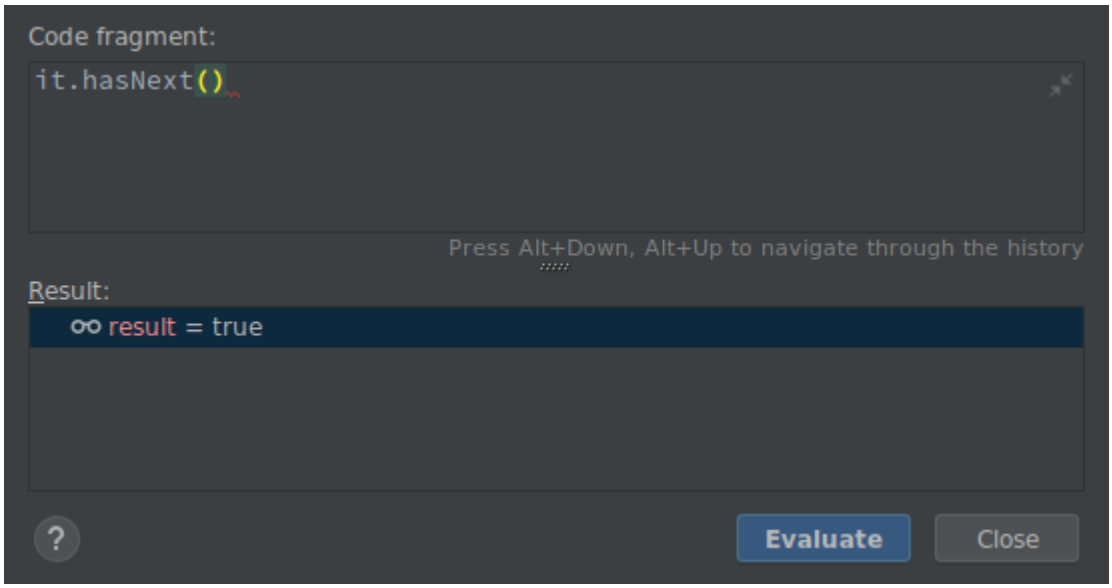

Volte na janela que mostrava o teste que falho. Clique com o botão direito no teste e selecione a opção "Debug testWellBehaved..." (no eclipse é apenas debug). Agora você entrou no modo debug:




O debugger permite que você acompanhe a execução do programa passo a passo e inspecione o valor de cada variável. Nesse caso queremos entrar na funçã takeOXO, então precisamos usar o comando Step Into (F7 no IntelliJ). Isso me levará até a implementação do método takeOXO(), onde eu posso ir pressionando Step over (F8 -- ao invés de entrar nas funções chamadas, simplesmente executa a linha e avança para a pórixma) até chegar no while:

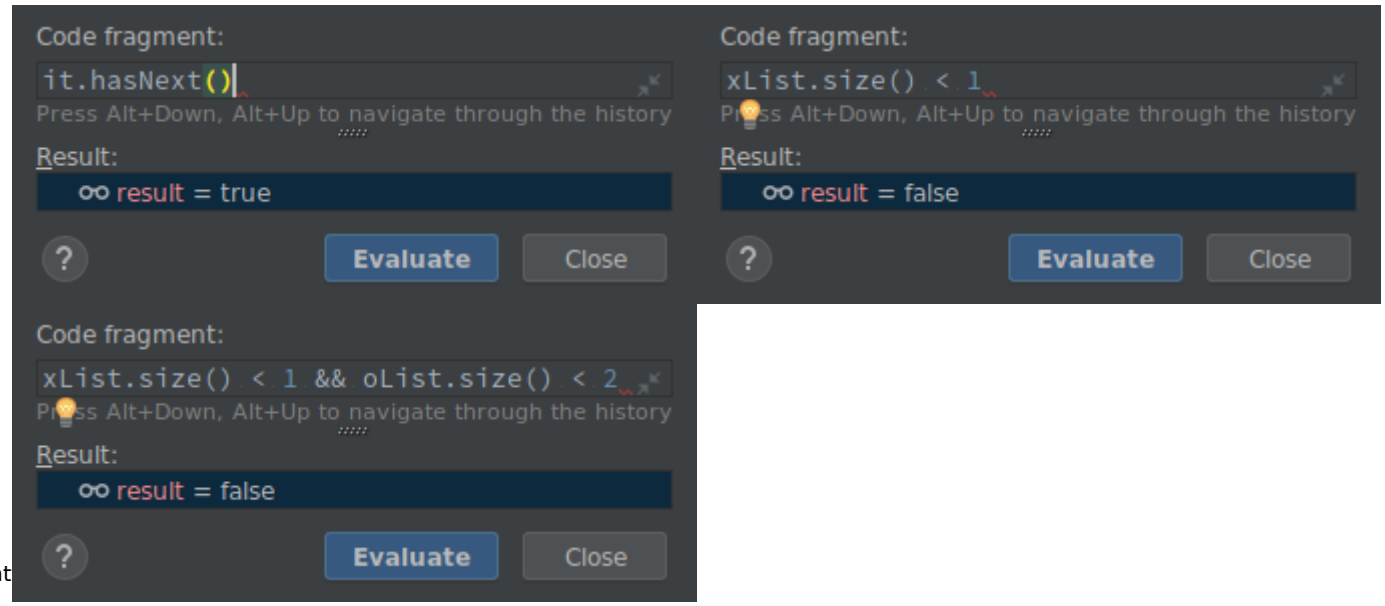
```
Iterator<Piece> it = queue.iterator();  it: LinkedList$ListItr@71
while (it.hasNext() && xList.size() < 1 && oList.size() < 2) {
    Piece piece = it.next();
    if (piece.getType() == X && xList.size() < 1) {
        xList.add(piece);
        it.remove();
        --nX;
    } else if (piece.getType() == O && oList.size() < 2) {
        oList.add(piece);
        it.remove();
        --nO;
    }
}
notifyAll();
```

A condição do while não é trivial, então é interessante inspecionar qual é o resultado da expressão it.hasNext(). No IntelliJ é possível acessar uma janela fabulosa apertando Alt+F8 (Run > Evaluate Expression). Essa janela permite que eu digite (quase) qualquer código Java e inspecione o resultado:




Usando F8 (step over) eu posso acompanhar a execução desse while. Na primeira iteração, vai ser adiiocnada uma peça O em oList. Na segunda iteração será adicionada uma peça X, e a terceira iteração não vai existir.

Frustado, eu posso reiniciar o debug e rodar até o momento em que ocorre o teste da condição do while para entrar na terceira iteração. Usando a Janela de Evaluate Expression, posso observar algumas coisas (da esquerda pra direita):




ht

Agora sim chegamos ao momento de apoiar o queixo no punho e encarar o monitor. Após alguns segundos, vocês conseguiriam perceber que

```
it.hasNext() && xList.size() < 1 && oList.size() < 2
```

deveria ser

```
it.hasNext() && (xList.size() < 1 || oList.size() < 2)
```

Última atualização: segunda, 5 Nov 2018, 23:23