CAPÍTULO 2

O modelo da bolsa de valores, datas e objetos imutáveis

"Primeiro aprenda ciência da computação e toda a teoria. Depois desenvolva um estilo de programação. E aí esqueça tudo e apenas 'hackeie'."
— George Carrette

O objetivo do FJ-22 é aprender boas práticas da orientação a objetos, do design de classes, uso correto dos design patterns, princípios de práticas ágeis de programação e a importância dos testes de unidade.

Dois livros que são seminais na área serão referenciados por diversas vezes pelo instrutor e pelo material: *Effective Java*, do Joshua Bloch, e *Design Patterns: Elements of Reusable Object-Oriented Software*, de Erich Gamma e outros (conhecido Gang of Four).

2.1 - A bolsa de valores

Poucas atividades humanas exercem tanto fascínio quanto o mercado de ações, assunto abordado exaustivamente em filmes, livros e em toda a cultura contemporânea. Somente em novembro de 2007, o total movimentado pela BOVESPA foi de R$ 128,7 bilhões. Destes, o volume movimentado por aplicações home *broker* foi de R$ 22,2 bilhões.

Neste curso, abordaremos esse assunto que, hoje em dia, chega a ser cotidiano desenvolvendo uma aplicação que interpreta os dados de um XML, trata e modela eles em Java e mostra gráficos pertinentes.

2.2 - Candlesticks: O Japão e o arroz

Yodoya Keian era um mercador japonês do século 17. Ele se tornou rapidamente muito rico, dadas as suas habilidades de transporte e precificação do arroz, uma mercadoria em crescente produção em consumo no país. Sua situação social de mercador não permitia que ele fosse tão rico dado o sistema de castas da época e, logo, o governo confiscou todo seu dinheiro e suas posses. Depois dele, outros vieram e tentaram esconder suas origens como mercadores: muitos tiveram seus filhos executados e seu dinheiro confiscado.

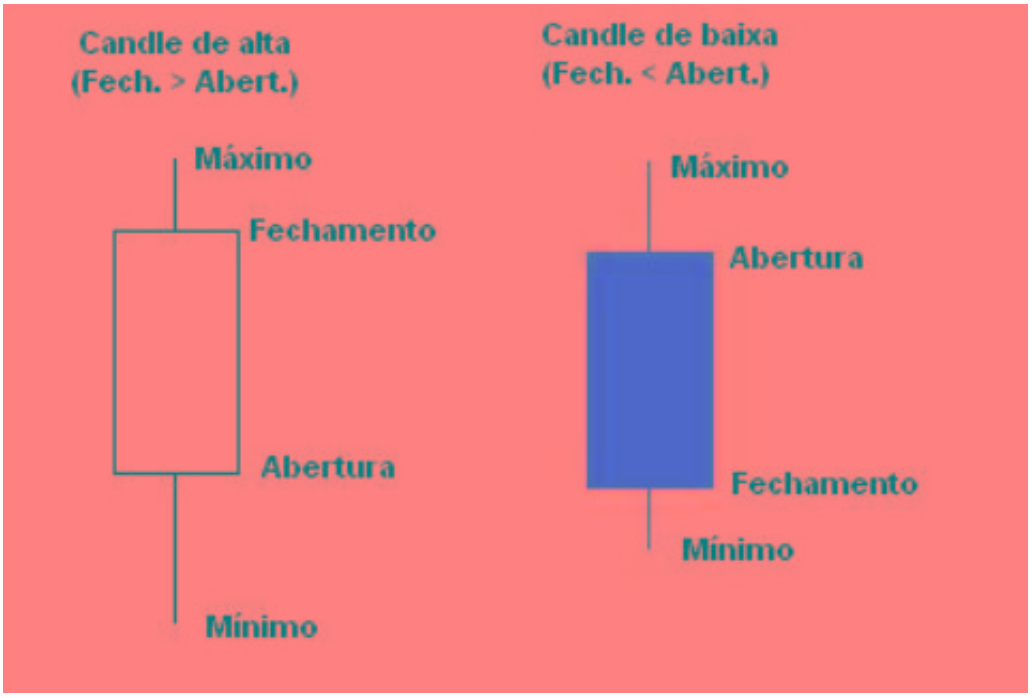
Apesar da triste história, foi em Dojima, no jardim do próprio Yodoya Keian, que nasceu a bolsa de arroz do Japão. Lá eram negociados, precificados e categorizados vários tipos de arroz. Para anotar os preços do arroz, desenhava-se figuras no papel. Essas figuras parecem muito com velas -- daí a analogia **candlestick**.

Esses desenhos eram feitos em um papel feito de... arroz! Apesar de usado a séculos, o mercado ocidental só se interessou pela técnica dos candlesticks recentemente, no último quarto de século.

Um candlestick indica 4 valores: o maior preço do dia, o menor preço do dia (as pontas), o primeiro preço do dia e o último preço do dia (conhecidos como abertura e fechamento, respectivamente).

Os preços de abertura e fechamento são as linhas horizontais e dependem do tipo de candle: se for de alta, o preço de abertura é embaixo; se for de baixa, é em cima. Um candle de alta costuma ter cor azul ou branca e os de baixa costumam ser vermelhos ou pretos. Caso o preço não tenha se movimentado, o candle tem a mesma cor que a do dia anterior.

Para calcular as informações necessárias para a construção de um `Candlestick`, são necessários os dados de todos as **negociações** (*trades*) de um dia. Uma **Negociação** possui três informações: o **preço** pelo qual foi comprado, a **quantidade** de ativos e a **data** em que ele foi executado.



Você pode ler mais sobre a história dos candles em: http://www.candlestickforum.com/PPF/Parameters/1_279_/candlestick.asp

Apesar de falarmos que o `Candlestick` representa os principais valores de *um dia*, ele pode ser usado para os mais variados intervalos de tempo: um candlestick pode representar 15 minutos, ou uma semana, dependendo se você está analisando o ativo para curto, médio ou longo prazo.

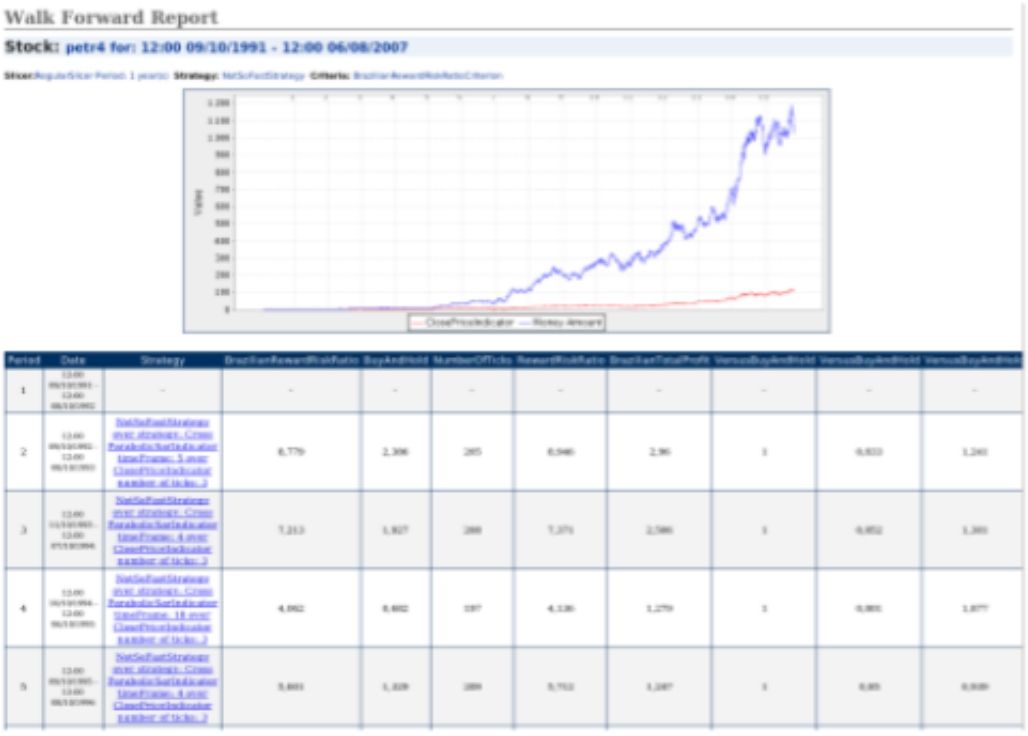
2.3 - O projeto Tail

A ideia do projeto **Tail** (**T**echnical **A**nalysis **I**ndicator **L**ibrary) nasceu quando um grupo de alunos da Universidade de São Paulo procurou o professor doutor Alfredo Goldman para orientá-los no desenvolvimento de um software para o projeto de conclusão de curso.

Ele então teve a ideia de juntar ao grupo alguns alunos do mestrado através de um sistema de coorientação, onde os mestrandos auxiliariam os graduandos na implementação, modelagem e metodologia do projeto. Somente então o grupo definiu o tema: o desenvolvimento de um software *open source* de análise técnica grafista (veremos o que é a análise técnica em capítulos posteriores).

O software está disponível no SourceForge:

```
http://sourceforge.net/projects/tail/
```



Essa ideia, ainda vaga, foi gradativamente tomando a forma do projeto desenvolvido. O grupo se reunia semanalmente adaptando o projeto, atribuindo novas tarefas e objetivos. Os graduandos tiveram a oportunidade de trabalhar em

conjunto com os mestrandos, que compartilharam suas experiências anteriores.

Objetivos do projeto Tail:

- Implementar os componentes básicos da análise técnica grafista: série temporal, operações de compra e venda e indicadores técnicos;
- Implementar as estratégias de compra e venda mais utilizadas no mercado, assim como permitir o rápido desenvolvimento de novas estratégias;
- Implementar um algoritmo genérico para determinar um momento apropriado de compra e venda de um ativo, através da escolha da melhor estratégia aplicada a uma série temporal;
- Permitir que o critério de escolha da melhor estratégia seja trocado e desenvolvido facilmente;
- Criar relatórios que facilitem o estudo e a compreensão dos resultados obtidos pelo algoritmo;
- Criar uma interface gráfica, permitindo o uso das ferramentas implementadas de forma fácil, rápida e de simples entendimento, mas que não limite os recursos da biblioteca;
- Arquitetura orientada a objetos, com o objetivo de ser facilmente escalável e de simples entendimento;
- Utilizar práticas de XP, adaptando-as conforme as necessidades do grupo.
- Manter a cobertura de testes superior a 90%;
- Analisar o funcionamento do sistema de coorientação, com o objetivo estendê-lo para projetos futuros.

O Tail foi desenvolvido por Alexandre Oki Takinami, Carlos Eduardo Mansur, Márcio Vinicius dos Santos, Thiago Garutti Thies, Paulo Silveira (mestre em Geometria Computacional pela USP e diretor da Caelum), Julian Monteiro (mestre em sistemas distribuídos pela USP e doutor pelo INRIA, em Sophia Antipolis, França) e Danilo Sato (mestre em Metodologias Ágeis pela USP e Lead Consultant na ThoughtWorks).

Esse projeto foi a primeira parceria entre a Caelum e a USP, onde a Caelum patrocinou o trabalho de conclusão de curso dos 4 graduandos, hoje todos formados.

Caso tenha curiosidade você pode acessar o CVS do projeto, utilizando o seguinte repositório:

```
http://tail.cvs.sourceforge.net/viewvc/tail/
```

2.4 - O projeto Argentum: modelando o sistema

O projeto Tail é bastante ambicioso. Tem centenas de recursos, em especial o de sugestão de quando comprar e de quando vender ações. O interessante durante o desenvolvimento do projeto Tail foi que muitos dos bons princípios de orientação a objetos, engenharia de software, design patterns e Programação eXtrema se encaixaram muito bem - por isso, nos inspiramos fortemente nele como base para o FJ-22.

Queremos modelar diversos objetos do nosso sistema, entre eles teremos:

- `Negociação` - guardando preço, quantidade e data;
- `Candlestick` - guardando as informações do Candle, além do volume de dinheiro negociado;
- `SerieTemporal` - que guarda um conjunto de candles.

Essas classes formarão a base do projeto que criaremos durante o treinamento, o **Argentum** (do latim, dinheiro ou prata). As funcionalidades do sistema serão as seguintes:

- Resumir `Negociacoes` em `Candlestick`s.Nossa base serão as negociações. Precisamos converter uma lista de negociações em uma lista de Candles.
- Converter `Candlestick`s em `SerieTemporal`.Dada uma lista de Candle, precisamos criar uma série temporal.
- Utilizar indicadores técnicosPara isso, implementar um pequeno *framework* de indicadores e criar alguns deles de forma a facilitar o desenvolvimento de novos.
- Gerar gráficosEmbutíveis e interativos na interface gráfica em Java, dos indicadores que criamos.

Para começar a modelar nosso sistema, precisamos entender alguns recursos de design de classes que ainda não foram discutidos no FJ-11. Entre eles podemos citar o uso da imutabilidade de objetos, uso de anotações e aprender a trabalhar e manipular datas usando a API do Java.

2.5 - Trabalhando com dinheiro

Até agora, não paramos muito para pensar nos tipos das nossas variáveis e já ganhamos o costume de automaticamente atribuir valores a variáveis `double`. Essa é, contudo, uma prática bastante perigosa!

O problema do `double` é que não é possível especificar a precisão mínima que ele vai guardar e, dessa forma, estamos sujeitos a problemas de arredondamento ao fracionar valores e voltar a somá-los. Por exemplo:

```
double cem = 100.0;
double tres = 3.0;
double resultado = cem / tres;

System.out.println(resultado);
    //    33.333?
    //    33.333333?
    //     33.3?
```

Se não queremos correr o risco de acontecer um arredondamento sem que percebamos, a alternativa é usar a classe `BigDecimal`, que lança exceção quando tentamos fazer uma operação cujo resultado é inexato.

Leia mais sobre ela na própria documentação do Java.

2.6 - Palavra chave final

A palavra chave `final` tem várias utilidades. Em uma classe, define que a classe nunca poderá ter uma filha, isso é, não pode ser estendida. A classe `String`, por exemplo, é `final`.

Como modificador de método, `final` indica que aquele método não pode ser reescrito. Métodos muito importantes costumam ser definidos assim. Claro que isso não é necessário declarar caso sua classe já seja `final`.

Ao usarmos como modificador na declaração de variável, indica que o valor daquela variável nunca poderá ser alterado, uma vez atribuído. Se a variável for um atributo, você tem que inicializar seu valor durante a construção do objeto - caso contrário, ocorre um erro de compilação, pois atributos `final` não são inicializados com valores default.

Imagine que, quando criamos um objeto `Negociacao`, não queremos que seu `valor` seja modificado:

```
class Negociacao {

  private final double valor;

  // getters e setters?

}
```

Esse código não compila, nem mesmo com um setter, pois o valor final deveria já ter sido inicializado. Para resolver isso, ou declaramos o `valor` da `Negociacao` direto na declaração do atributo (o que não faz muito sentido nesse caso), ou então populamos pelo construtor:

```
class Negociacao {

  private final double valor;

  public Negociacao(double valor) {
    this.valor = valor;
  }

  // podemos ter um getter, mas nao um setter aqui!

}
```

Uma variável `static final` tem uma cara de constante daquela classe e, se for `public static final`, aí parece uma constante global! Por exemplo, na classe `Collections` do `java.util` existe uma constante `public static final` chamada `EMPTY_LIST`. É convenção que constantes sejam declaradas letras maiúsculas e separadas por travessão (*underscore*) em vez de usar o padrão *camel case*. Outros bons exemplos são o `PI` e o `E`, dentro da `java.lang.Math`.

Isso é muito utilizado, mas hoje no java 5 para criarmos constantes costuma ser muito mais interessante utilizarmos o recurso de enumerações que, além de

tipadas, já possuem diversos métodos auxiliares.

No caso da classe `Negociacao`, no entanto, bastará usarmos atributos finais e também marcarmos a própria classe como final para que ela crie apenas objetos imutáveis.

2.7 - Imutabilidade de objetos

> **Effective Java**
>
> Item 15: Minimize mutabilidade

Para que uma classe seja imutável, ela precisa ter algumas características:

- Nenhum método pode modificar seu estado;
- A classe deve ser `final`;
- Os atributos devem ser privados;
- Os atributos devem ser `final`, apenas para legibilidade de código, já que não há métodos que modificam o estado do objeto;
- Caso sua classe tenha composições com objetos mutáveis, eles devem ter acesso exclusivo pela sua classe.

Diversas classes no Java são imutáveis, como a `String` e todas as classes *wrapper*. Outro excelente exemplo de imutabilidade são as classes `BigInteger` e `BigDecimal`:

Qual seria a motivação de criar uma classe de tal maneira?

Objetos podem compartilhar suas composições

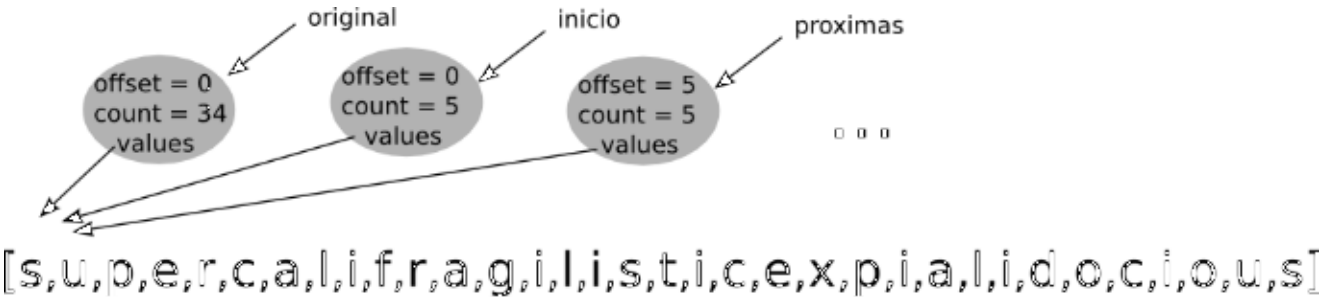
Como o objeto é imutável, a composição interna de cada um pode ser compartilhada entre eles, já que não há chances de algum deles mudar tais atributos. Esse compartilhamento educado possibilita fazer *cache* de suas partes internas, além de facilitar a manipulação desses objetos.

Isso pode ser encarado como o famoso design pattern **Flyweight**.

É fácil entender os benefícios dessa prática quando olhamos para o caso da `String`: objetos do tipo `String` que contêm exatamente o mesmo texto ou partes exatas do texto original (como no caso de usarmos o `substring`) compartilham a *array* privada de `chars`!

Na prática, o que isso quer dizer é que se você tem uma String muito longa e cria várias outras com trechos da original, você não terá que armazenar os caracteres de novo para cada trecho: eles utilizarão o `array` de `chars` da `String` original!

```
String palavra = "supercalifragilisticexpialidocious";
String inicio = palavra.substring(0, 5);
String proximas = palavra.substring(5, 10);
String outras = palavra.substring(10, 15);
String resto = palavra.substring(15);
```



Esses objetos também são ideais para usar como chave de tabelas de hash.

Thread safety

Uma das principais vantagens da imutabilidade é em relação a concorrência. Simplesmente não precisamos nos preocupar em relação a isso: como não há método que mude o estado do objeto, então não há como fazer duas modificações acontecerem concorrentemente!

Objetos mais simples

Uma classe imutável é mais simples de dar manutenção. Como não há chances de seu objeto ser modificado, você tem uma série de garantias sobre o uso daquela classe.

Se os construtores já abrangem todas as regras necessárias para validar o estado do objeto, não há preocupação em relação a manter o estado consistente, já que não há chances de modificação.

Uma boa prática de programação é evitar tocar em variáveis parâmetros de um método. Com objetos imutáveis nem existe esse risco quando você os recebe como parâmetro.

Se nossa classe `Negociacao` é imutável, isso remove muitas dúvidas e medos que poderíamos ter durante o desenvolvimento do nosso projeto: saberemos em todos os pontos que os valores da negociação são sempre os mesmos, não corremos o risco de um método que constrói o `candlestick` mexer nos nossos atributos (deixando ou não num estado inconsistente), além de a imutabilidade também garantir que não haverá problemas no caso de acesso concorrente ao objeto.

2.8 - Trabalhando com datas: Date e Calendar

Se você fez o FJ-21 conosco, já teve que lidar com as conversões entre `Date` e `Calendar` para pegar a entrada de data de um texto digitado pelo usuário e convertê-lo para um objeto que representa datas em Java.

A classe mais antiga que representa uma data dentro do Java é a `Date`. Ela armazena a data de forma cada momento do tempo seja representado por um número - isso quer dizer, que o `Date` guarda todas as datas como milissegundos que se passaram desde 01/01/1970.

O armazenamento dessa forma não é de todo ruim, mas o problema é que a API não traz métodos que ajudem muito a lidar com situações do dia como, por exemplo, adicionar dias ou meses a uma data.

A classe `Date` não mais é recomendada porque a maior parte de seus métodos estão marcados como `deprecated`, porém ela tem amplo uso legado nas bibliotecas do java. Ela foi substituída no Java 1.1 pelo `Calendar`, para haver suporte correto à internacionalização e à localização do sistema de datas.

Calendar: evolução do Date

A classe abstrata `Calendar` também encapsula um instante em milissegundos, como a `Date`, mas ela provê métodos para manipulação desse momento em termos mais cotidianos como dias, meses e anos. Por ser abstrata, no entanto, não podemos criar objetos que são simplesmente `Calendar`s.

A subclasse concreta de `Calendar` mais usada é a `GregorianCalendar`, que representa o calendário usado pela maior parte dos países -- outras implementações existem, como a do calendário budista `BuddhistCalendar`, mas estas são bem menos usadas e devolvidas de acordo com seu `Locale`.

Há ainda a API nova do Java 8, chamada `java.time`, desenvolvida com base no

Joda Time. Trabalharemos aqui com Calendar por ser ainda muito mais difundida, dado que o Java 8 é muito recente. Caso você tenha a possibilidade de trabalhar com Java 8, favoreça o uso da API nova.

Para obter um `Calendar` que encapsula o instante atual (data e hora), usamos o método estático `getInstance()` de `Calendar`.

```
Calendar agora = Calendar.getInstance();
```

Porque não damos `new` diretamente em `GregorianCalendar`? A API do Java fornece esse método estático que **fabrica** um objeto `Calendar` de acordo com uma série de regras que estão encapsuladas dentro de `getInstance`. Esse é o padrão de projeto *factory*, que utilizamos quando queremos esconder a maneira em que um objeto é instanciado. Dessa maneira podemos trocar implementações devolvidas como retorno a medida que nossas necessidades mudem.

Nesse caso algum país que use calendários diferente do gregoriano pode implementar esse método de maneira adequada, retornando o que for necessário de acordo com o `Locale` configurado na máquina.

> **Effective Java**
>
> Item 1: Considere utilizar Factory com métodos estáticos em vez de construtores

Repare ainda que há uma sobrecarga desse método que recebe `Locale` ou `Timezone` como argumento, caso você queira que essa *factory* trabalhe com valores diferentes dos valores que a JVM descobrir em relação ao seu ambiente.

Um outro excelente exemplo de *factory* é o `DriverManager` do `java.sql` que fabrica `Connection` de acordo com os argumentos passados.

A partir de um `Calendar`, podemos saber o valor de seus campos, como ano, mês, dia, hora, minuto, etc. Para isso, usamos o método `get` que recebe um inteiro representando o campo; os valores possíveis estão em constantes na classe `Calendar`.

No exemplo abaixo, imprimimos o dia de hoje e o dia da semana correspondente. Note que o dia da semana devolvido é um inteiro que representa o dia da semana (`Calendar.MONDAY` etc):

```
Calendar c = Calendar.getInstance();
System.out.println("Dia do Mês: " + c.get(Calendar.DAY_OF_MONTH));
System.out.println("Dia da Semana: " + c.get(Calendar.DAY_OF_WEEK));
```

Um possível resultado é:

```
Dia do Mês: 4
Dia da Semana: 5
```

No exemplo acima, o dia da semana **5** representa a **quinta-feira**.

Da mesma forma que podemos pegar os valores dos campos, podemos atribuir novos valores a esses campos por meio dos métodos `set`.

Há diversos métodos `set` em `Calendar`. O mais geral é o que recebe dois argumentos: o primeiro indica qual é o campo (usando aquelas constantes de `Calendar`) e, o segundo, o novo valor. Além desse método, outros métodos `set` recebem valores de determinados campos; o `set` de três argumentos, por exemplo, recebe ano, mês e dia. Vejamos um exemplo de como alterar a data de hoje:

```
Calendar c = Calendar.getInstance();
c.set(2011, Calendar.DECEMBER, 25, 10, 30);
// mudamos a data para as 10:30am do Natal
```

Outro método bastante usado é `add`, que adiciona uma certa quantidade a qualquer campo do `Calendar`. Por exemplo, para uma aplicação de agenda, queremos adicionar um ano à data de hoje:

```
Calendar c = Calendar.getInstance();
c.add(Calendar.YEAR, 1); // adiciona 1 ao ano
```

Note que, embora o método se chame `add`, você pode usá-lo para subtrair valores também; basta colocar uma quantidade negativa no segundo argumento.

Os métodos `after` e `before` são usados para comparar o objeto `Calendar` em questão a outro `Calendar`. O método `after` devolverá `true` quando o objeto atual do `Calendar` representar um momento posterior ao do `Calendar` passado como argumento. Por exemplo, `after` devolverá `false` se compararmos o dia das crianças com o Natal, pois o dia das crianças não vem depois do Natal:

```
Calendar c1 = new GregorianCalendar(2005, Calendar.OCTOBER, 12);
Calendar c2 = new GregorianCalendar(2005, Calendar.DECEMBER, 25);
System.out.println(c1.after(c2));
```

Analogamente, o método `before` verifica se o momento em questão vem antes

do momento do `Calendar` que foi passado como argumento. No exemplo acima, `c1.before(c2)` devolverá `true`, pois o dia das crianças vem antes do Natal.

Note que `Calendar` implementa `Comparable`. Isso quer dizer que você pode usar o método `compareTo` para comparar dois calendários. No fundo, `after` e `before` usam o `compareTo` para dar suas respostas - apenas, fazem tal comparação de uma forma mais elegante e encapsulada.

Por último, um dos problemas mais comuns quando lidamos com datas é verificar o intervalo de dias entre duas datas que podem ser até de anos diferentes. O método abaixo devolve o número de dias entre dois objetos Calendar. O cálculo é feito pegando a diferença entre as datas em milissegundos e dividindo esse valor pelo número de milissegundos em um dia:

```
public int diferencaEmDias(Calendar c1, Calendar c2) {
  long m1 = c1.getTimeInMillis();
  long m2 = c2.getTimeInMillis();
  return (int) ((m2 - m1) / (24*60*60*1000));
}
```

Relacionando Date e Calendar

Você pode pegar um `Date` de um `Calendar` e vice-versa através dos métodos `getTime` e `setTime` presentes na classe `Calendar`:

```
Calendar c = new GregorianCalendar(2005, Calendar.OCTOBER, 12);
Date d = c.getTime();
c.setTime(d);
```

Isso faz com que você possa operar com datas da maneira nova, mesmo que as APIs ainda usem objetos do tipo `Date` (como é o caso de `java.sql`).

Para saber mais: Classes Deprecated e o JodaTime

O que fazer quando descobrimos que algum método ou alguma classe não saiu bem do jeito que deveria? Simplesmente apagá-la e criar uma nova?

Essa é uma alternativa possível quando apenas o seu programa usa tal classe, mas definitivamente não é uma boa alternativa se sua classe já foi usada por milhões de pessoas no mundo todo.

É o caso das classes do Java. Algumas delas (`Date`, por exemplo) são repensadas anos depois de serem lançadas e soluções melhores aparecem (`Calendar`). Mas, para não quebrar compatibilidade com códigos existentes, o Java mantém as

funcionalidades problemáticas ainda na plataforma, mesmo que uma solução melhor exista.

Mas como desencorajar códigos novos a usarem funcionalidades antigas e não mais recomendadas? A prática no Java para isso é marcá-las como **deprecated**. Isso indica aos programadores que não devemos mais usá-las e que futuramente, em uma versão mais nova do Java, podem sair da API (embora isso nunca tenha ocorrido na prática).

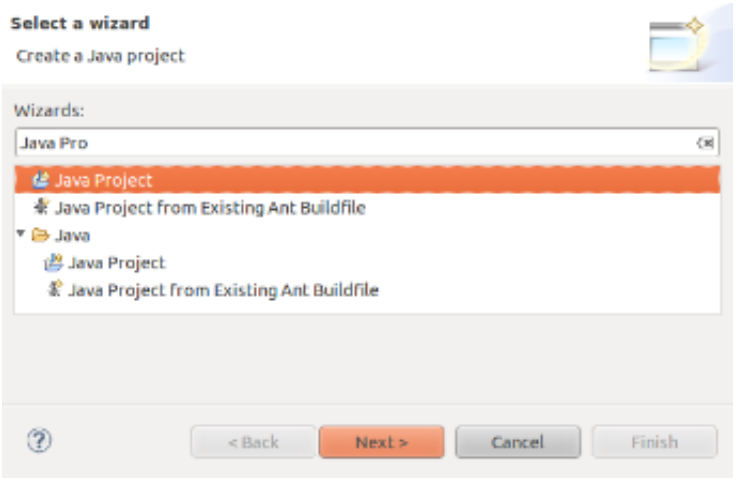
Antes do Java 5, para falar que algo era deprecated, usava-se um comentário especial no Javadoc. A partir do Java 5, a anotação `@Deprecated` foi adicionada à plataforma e garante verificações do próprio compilador (que gera um warning). Olhe o Javadoc da classe `Date` para ver tudo que foi deprecated.

A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Numa próxima versão novas classes para facilitar ainda mais o trabalho com datas e horários devem entrar na especificação do Java, baseadas na excelente biblioteca **JodaTime**.

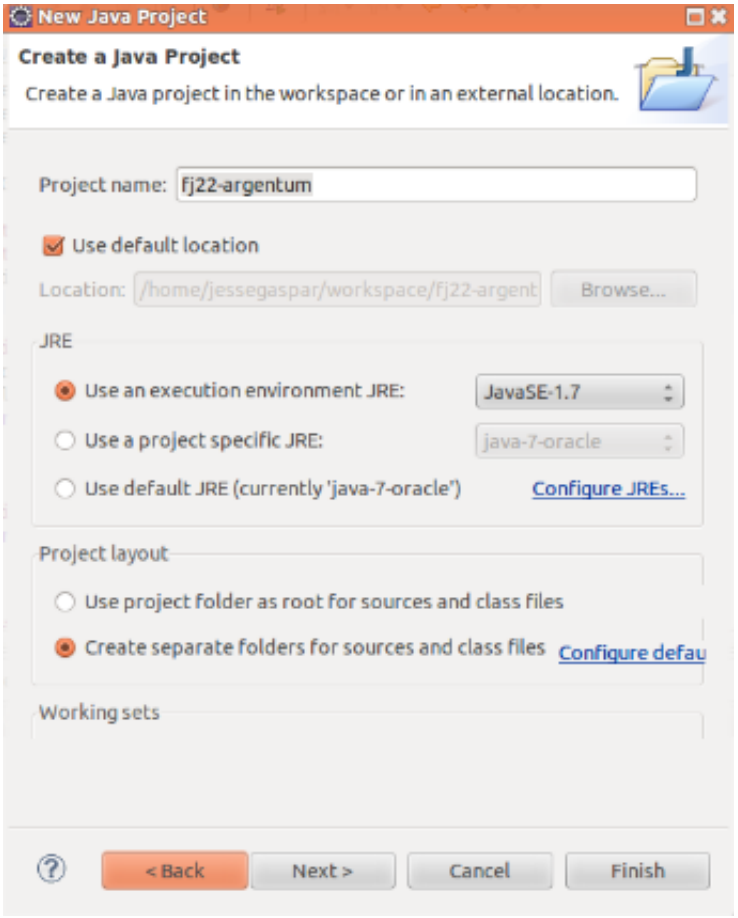
Para mais informações: http://blog.caelum.com.br/2007/03/15/jsr-310-date-and-time-api/ http://jcp.org/en/jsr/detail?id=310

2.9 - Exercícios: o modelo do Argentum

1. Vamos criar o projeto `fj22-argentum` no Eclipse, já com o foco em usar a IDE melhor: use o atalho **ctrl + N**, que *cria novo...* e comece a digitar *Java Project*:



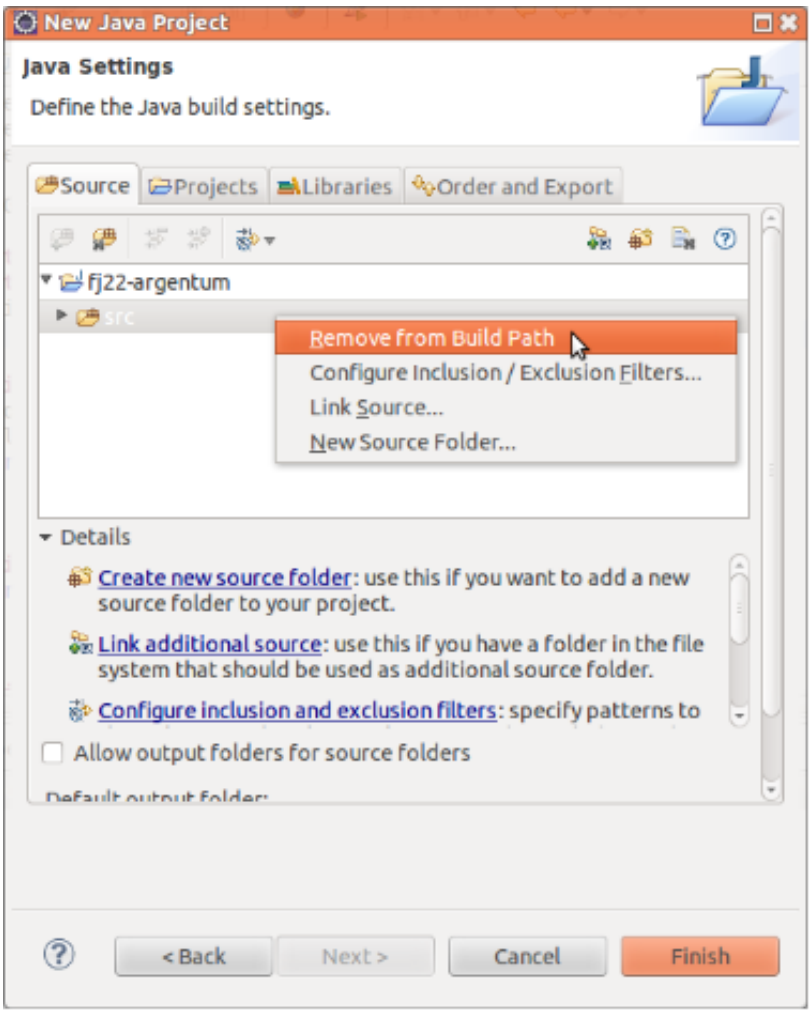
2. Na janela que abrirá em sequência, preencha o nome do projeto como **fj22-argentum** e clique em **Next**:



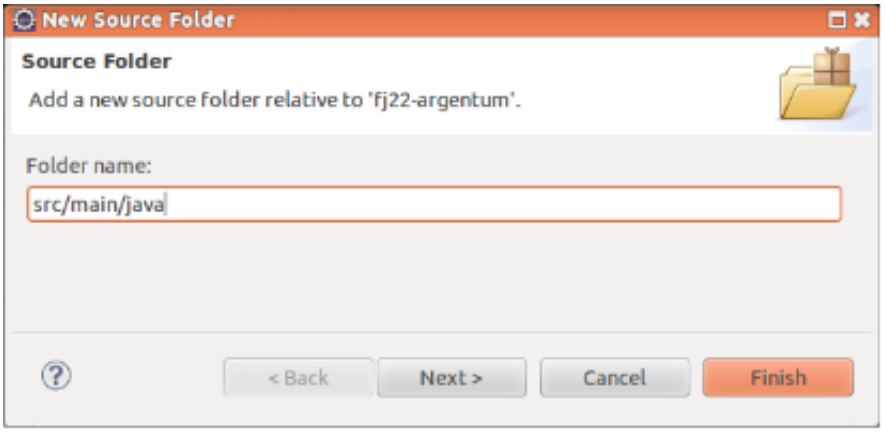
3. Na próxima tela, podemos definir uma série de configurações do projeto (que também podem ser feitas depois, através do menu *Build path -> Configure build path*, clicando com o botão da direita no projeto.

Queremos mudar o diretório que conterá nosso código fonte. Faremos isso para organizar melhor nosso projeto e utilizar convenções amplamente utilizadas no mercado.

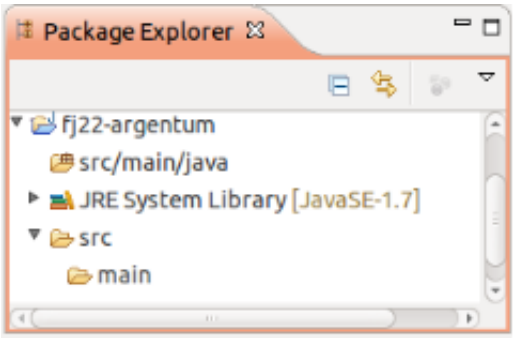
Nessa tela, **remova** o diretório `src` da lista de diretórios fonte:



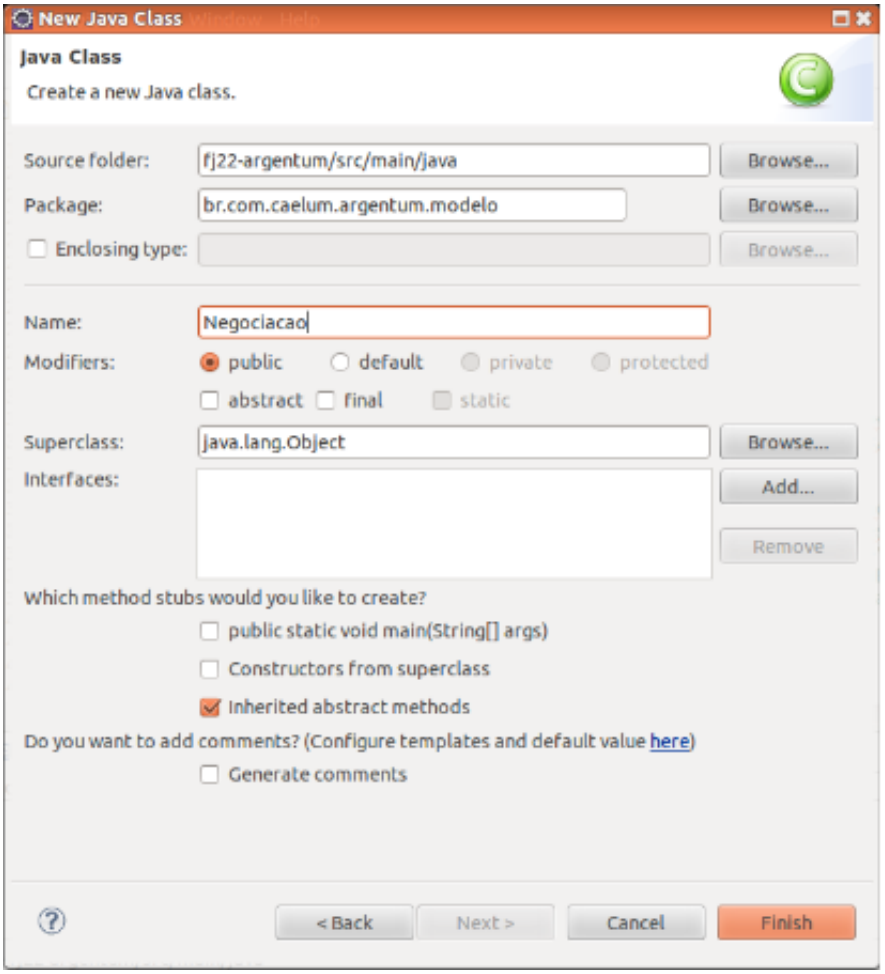
Agora, na mesma tela, adicione um novo diretório fonte, chamado **src/main/java**. Para isso, clique em **Create new source folder** e preencha com **src/main/java**:



4. Agora basta clicar em **Finish**. A estrutura final de seu projeto deve estar parecida com isso:



5. Crie a classe usando **ctrl + N** `Class`, chamada `Negociacao` e dentro do pacote `br.com.caelum.argentum.modelo`:



6. Transforme a classe em **final** e já declare os três atributos que fazem parte de uma negociação da bolsa de valores (também como final):

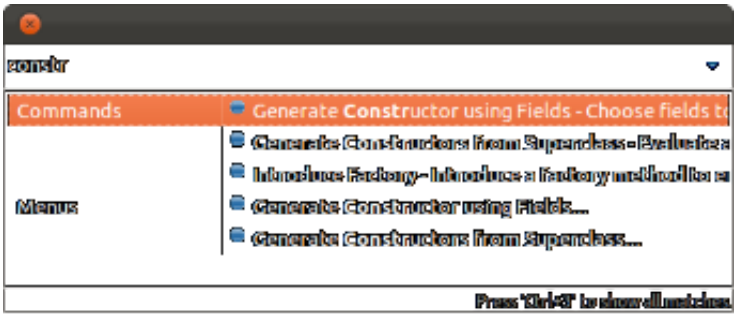
```
public final class Negociacao {
  private final double preco;
  private final int quantidade;
  private final Calendar data;

}
```

Não esqueça de importar o `Calendar`!

7. Vamos criar um construtor que recebe esses dados, já que são obrigatórios para nosso domínio. Em vez de fazer isso na mão, na edição da classe, use o atalho **ctrl**

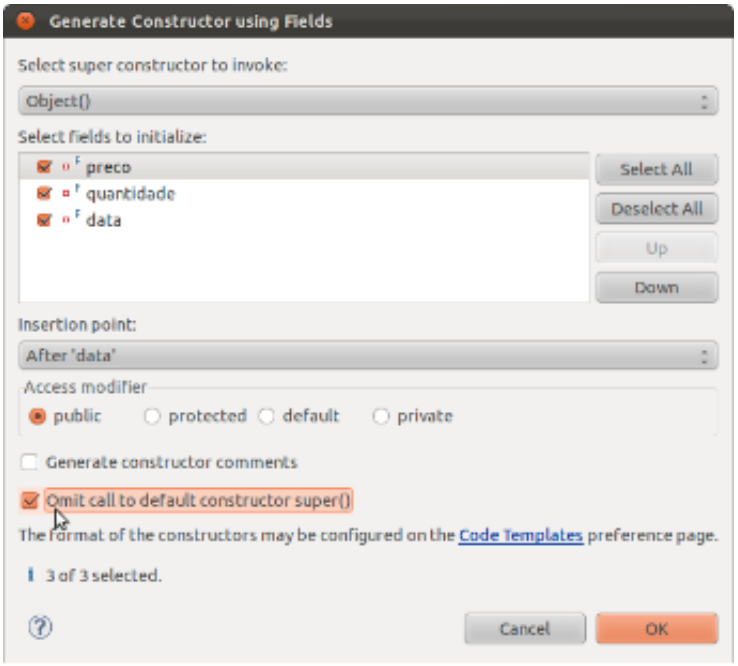
+ 3 e comece a digitar *constructor*. Ele vai mostrar uma lista das opções que contêm essa palavra: escolha a *Generate constructor using fields*.

Alternativamente, tecle **ctrl + 3** e digite *GCUF*, que são as iniciais do menu que queremos acessar.



Agora, selecione todos os campos e marque para omitir a invocação ao `super`, como na tela abaixo.

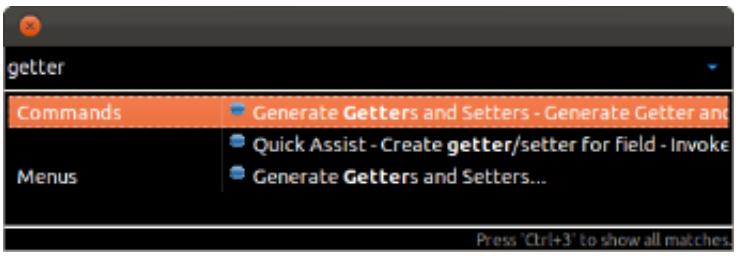
Atenção para deixar os campos na ordem 'preco, quantidade, data'. Você pode usar os botões *Up* e *Down* para mudar a ordem.



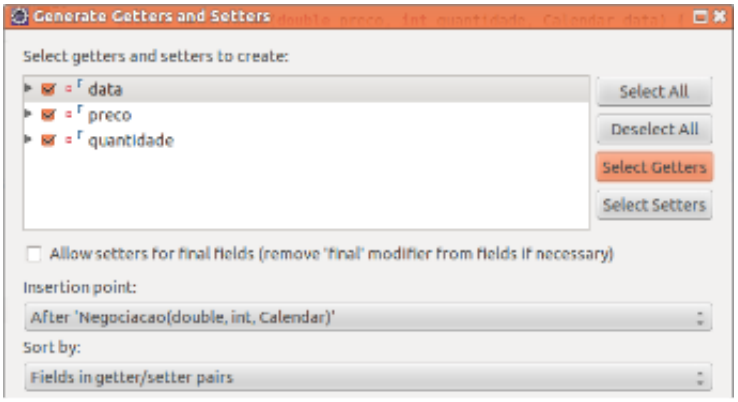
Pronto! Mande gerar. O seguinte código que será gerado:

```
public Negociacao(double preco, int quantidade, Calendar data) {
  this.preco = preco;
  this.quantidade = quantidade;
  this.data = data;
}
```

8. Agora, vamos gerar os getters dessa classe. Faça **ctrl + 3** e comece a digitar *getter*, as opções aparecerão e basta você escolher *generate getters and setters*. É sempre bom praticar os atalhos do **ctrl + 3**.



Selecione os getters e depois **Finish**:



9. Verifique sua classe. Ela deve estar assim:

```
public final class Negociacao {

  private final double preco;
  private final int quantidade;
  private final Calendar data;

  public Negociacao(double preco, int quantidade, Calendar data) {
    this.preco = preco;
    this.quantidade = quantidade;
    this.data = data;
  }

  public double getPreco() {
    return preco;
  }

  public int getQuantidade() {
    return quantidade;
  }

  public Calendar getData() {
    return data;
  }
}
```

10. Um dado importante para termos noção da estabilidade de uma ação na bolsa de valores é o volume de dinheiro negociado em um período.

Vamos fazer nossa classe `Negociacao` devolver o volume de dinheiro transferido naquela negociação. Na prática, é só multiplicar o `preco` pago pela `quantidade` de ações negociadas, resultando no total de dinheiro que aquela negociação realizou.

Adicione o método `getVolume` na classe `Negociacao`:

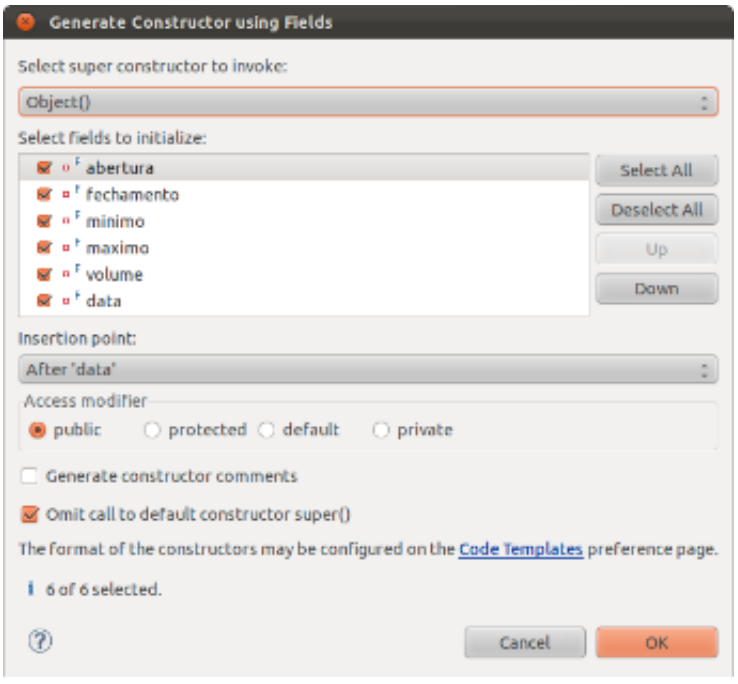
```
public double getVolume() {
  return preco * quantidade;
}
```

Repare que um método que parece ser um simples *getter* pode (e deve muitas vezes) encapsular regras de negócio e não necessariamente refletem um atributo da classe.

11. Siga o mesmo procedimento para criar a classe `Candlestick`. Use o **ctrl + N** *Class* para isso, marque-a como **final** e adicione os seguintes atributos **finais**, nessa ordem:

```
public final class Candlestick {
  private final double abertura;
  private final double fechamento;
  private final double minimo;
  private final double maximo;
  private final double volume;
  private final Calendar data;

}
```

12. Use o **ctrl + 3** para gerar o construtor com os seis atributos. Atenção à ordem dos parâmetros no construtor:



13. Gere também os seis respectivos getters, usando o **ctrl + 3**.

A classe final deve ficar parecida com a que segue:

```
public final class Candlestick {
  private final double abertura;
```

```
  private final double fechamento;
  private final double minimo;
  private final double maximo;
  private final double volume;
  private final Calendar data;

  public Candlestick(double abertura,double fechamento,double minimo,
      double maximo, double volume, Calendar data) {
    this.abertura = abertura;
    this.fechamento = fechamento;
    this.minimo = minimo;
    this.maximo = maximo;
    this.volume = volume;
    this.data = data;
  }

  public double getAbertura() {
    return abertura;
  }
  public double getFechamento() {
    return fechamento;
  }
  public double getMinimo() {
    return minimo;
  }
  public double getMaximo() {
    return maximo;
  }
  public double getVolume() {
    return volume;
  }
  public Calendar getData() {
    return data;
  }
}
```

14. (opcional) Vamos *adicionar* dois métodos de negócio, para que o `Candlestick` possa nos dizer se ele é do tipo de alta, ou se é de baixa:

```
public boolean isAlta() {
  return this.abertura < this.fechamento;
}

public boolean isBaixa() {
  return this.abertura > this.fechamento;
}
```

2.10 – Resumo diário das Negociações

Agora que temos as classes que representam negociações na bolsa de valores (`Negociacao`) e resumos diários dessas negociações (`Candlestick`), falta apenas fazer a ação de resumir as negociações de um dia em uma candle.

A regra é um tanto simples: dentre uma lista de negociações, precisamos descobrir quais são os valores a preencher na `Candlestick`:

- **Abertura**: preço da primeira negociação do dia;
- **Fechamento**: preço da última negociação do dia;
- **Mínimo**: preço da negociação mais barata do dia;
- **Máximo**: preço da negociação mais cara do dia;
- **Volume**: quantidade de dinheiro que passou em todas as negociações nesse dia;
- **Data**: a qual dia o resumo se refere.

Algumas dessas informações são fáceis de encontrar por que temos uma **convenção** no sistema: quando vamos criar a candle, a lista de negociações já vem ordenada por tempo. Dessa forma, a abertura e o fechamento são triviais: basta recuperar o preço , respectivamente, da primeira e da última negociações do dia!

Já mínimo, máximo e volume precisam que todos os valores sejam verificados. Dessa forma, precisamos passar por cada negociação da lista verificando se aquele valor é menor do que todos os outros que já vimos, maior que nosso máximo atual. Aproveitando esse processo de passar por cada negociação, já vamos somando o volume de cada negociação.

O algoritmo, agora, está completamente especificado! Basta passarmos essas ideias para código. Para isso, lembremos, você pode usar alguns atalhos que já vimos antes:

- **Ctrl + N**: cria novo(a)...
- **Ctrl + espaço**: autocompleta
- **Ctrl + 1**: resolve pequenos problemas de compilação e atribui objetos a variáveis.

Em que classe colocar?

Falta apenas, antes de pôr em prática o que aprendemos, decidirmos onde vai esse código de criação de Candlestick. Pense bem a respeito disso: será que uma negociação deveria saber resumir vários de si em uma candle? Ou será que uma Candlestick deveria saber gerar um objeto do próprio tipo Candlestick a partir de uma lista de negociações.

Em ambos os cenários, nossos modelos têm que ter informações a mais que, na

realidade, são responsabilidades que não cabem a eles!

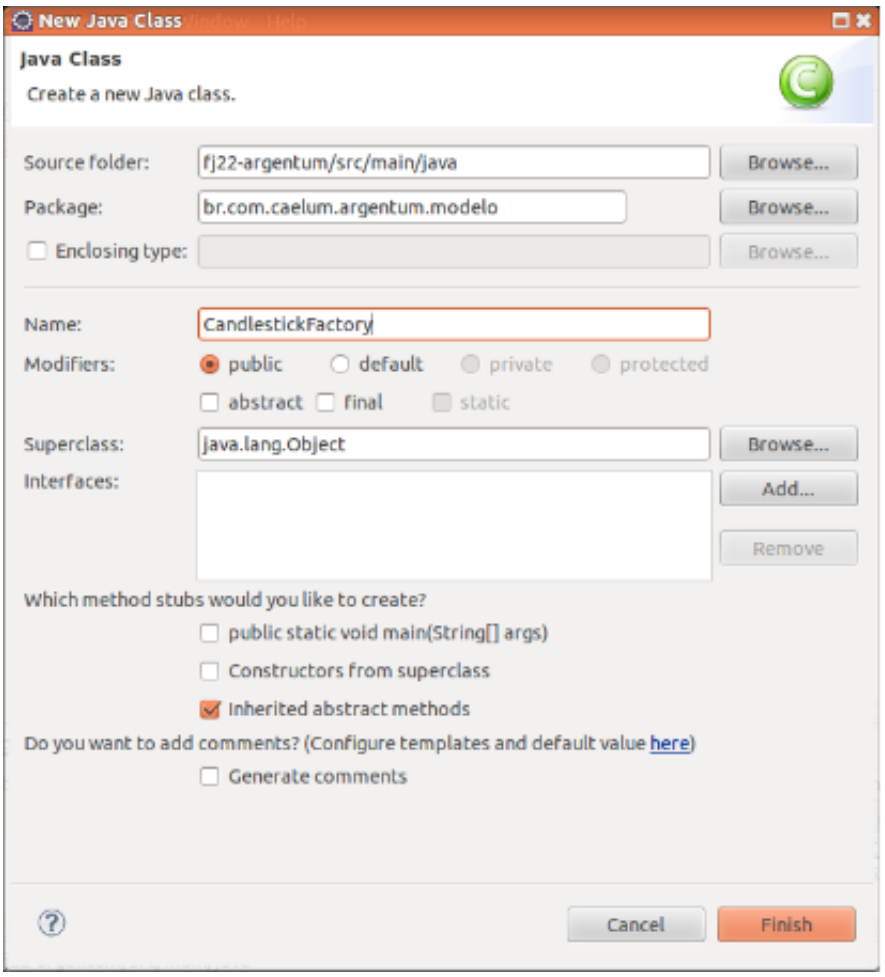
Criaremos, então, uma classe que: *dado a matéria-prima, nos constrói uma candle*. E uma classe com esse comportamento, que recebem o necessário para criar um objeto e **encapsulam** o algoritmo para tal criação, costuma ser chamadas de `Factory`.

No nosso caso particular, essa é uma fábrica que cria `Candlesticks`, então, seu nome fica `CandlestickFactory`.

Perceba que esse nome, apesar de ser um tal *Design Pattern* nada mais faz do que encapsular uma lógica um pouco mais complexa, isto é, apenas aplica boas práticas de orientação a objetos que você já vem estudando desde o FJ-11.

2.11 - Exercícios: fábrica de Candlestick

1. Como o resumo de `Negociacoes` em um `Candlestick` é um processo complicado, vamos encapsular sua construção através de uma fábrica, assim como vimos a classe `Calendar`, porém o método de fabricação ficará numa classe a parte, o que também é muito comum. Vamos criar a classe `CandlestickFactory` dentro do pacote `br.com.caelum.argentum.modelo`:



Depois de criá-la, adicione a assinatura do método `constroiCandleParaData` como abaixo:

```
public class CandlestickFactory {
  // ctrl + 1 para adicionar o return automaticamente
  public Candlestick constroiCandleParaData(Calendar data,
                    List<Negociacao> negociacoes) {

  }
}
```

2. Procuraremos os preços máximo e mínimo percorrendo todas as negociações. Para isso usaremos variáveis auxiliares `maximo` e `minimo` e, dentro do for, verificaremos se o preço da negociação atual é maior que o valor da variável maximo. Se não for, veremos se ele é menor que o `minimo`. Calcularemos o volume somando o volume de cada negociação em uma variável auxiliar chamada `volume`. Poderemos pegar o preço de abertura através de `negociacoes.get(0)` e o de fechamento por `negociacoes.get(negociacoes.size() - 1)`.

```
public class CandlestickFactory {
  public Candlestick constroiCandleParaData(Calendar data,
                    List<Negociacao> negociacoes) {
    double maximo = negociacoes.get(0).getPreco();
    double minimo = negociacoes.get(0).getPreco();
    double volume = 0;
```

```
    // digite foreach e dê um ctrl + espaço para ajudar a
    // criar o bloco abaixo!
    for (Negociacao negociacao : negociacoes) {
      volume += negociacao.getVolume();

      if (negociacao.getPreco() > maximo) {
        maximo = negociacao.getPreco();
      } else if (negociacao.getPreco() < minimo) {
        minimo = negociacao.getPreco();
      }
    }

    double abertura = negociacoes.get(0).getPreco();
    double fechamento = negociacoes.get(negociacoes.size()-1).getPreco();

    return new Candlestick(abertura, fechamento, minimo, maximo,
                           volume, data);
  }
}
```

3. Vamos testar nosso código, criando 4 negociações e calculando o `Candlestick`, finalmente. Crie a classe `TestaCandlestickFactory` no pacote `br.com.caelum.argentum.testes`

```
public class TestaCandlestickFactory {

  public static void main(String[] args) {
    Calendar hoje = Calendar.getInstance();

    Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
    Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
    Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
    Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

    List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
              negociacao3, negociacao4);

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(
                        hoje, negociacoes);

    System.out.println(candle.getAbertura());
    System.out.println(candle.getFechamento());
    System.out.println(candle.getMinimo());
    System.out.println(candle.getMaximo());
    System.out.println(candle.getVolume());
  }
}
```

O método `asList` da classe `java.util.Arrays` cria uma lista dada uma array. Mas não passamos nenhuma array como argumento! Isso acontece porque esse método aceita `varargs`, possibilitando que invoquemos esse método **separando a array por vírgula**. Algo parecido com um *autoboxing* de arrays.

Effective Java

Item 47: Conheça e use as bibliotecas!

A saída deve ser parecida com:

```
40.5
42.3
39.8
45.0
16760.0
```


2.12 - Exercícios opcionais

1. **Effective Java**

Item 10: Sempre reescreva o `toString`

Reescreva o `toString` da classe `Candlestick`. Como o `toString` da classe `Calendar` retorna uma `String` bastante complexa, faça com que a data seja corretamente visualizada, usando para isso o `SimpleDateFormat`. Procure sobre essa classe na API do Java.

Ao imprimir um candlestick, por exemplo, a saída deve ser algo como segue:

```
[Abertura 40.5, Fechamento 42.3, Mínima 39.8, Máxima 45.0,
Volume 145234.20, Data 12/07/2008]
```

Para reescrever um método e tirar proveito do Eclipse, a maneira mais direta é de dentro da classe `Candlestick`, fora de qualquer método, pressionar **ctrl + espaço**.

Uma lista com todas as opções de métodos que você pode reescrever vai aparecer. Escolha o `toString`, e ao pressionar *enter* o esqueleto da reescrita será montado.

2. Um `double` segue uma regra bem definida em relação a contas e arredondamento, e para ser rápido e caber em 64 bits, não tem precisão infinita. A classe `BigDecimal` pode oferecer recursos mais interessantes em um ambiente onde as casas decimais são valiosas, como um sistema financeiro. Pesquise a respeito.

3. O construtor da classe `Candlestick` é simplesmente **muito** grande. Poderíamos usar uma *factory*, porém continuaríamos passando muitos argumentos para um determinado método.

Quando construir um objeto é complicado, ou confuso, costumamos usar o padrão **Builder** para resolver isso. Builder é uma classe que ajuda você a construir um determinado objeto em uma série de passos, independente da ordem.

> **Effective Java**
>
> Item 2: Considere usar um builder se o construtor tiver muitos parâmetros!

A ideia é que possamos criar um *candle* da seguinte maneira:

```
CandleBuilder builder = new CandleBuilder();

builder.comAbertura(40.5);
builder.comFechamento(42.3);
builder.comMinimo(39.8);
builder.comMaximo(45.0);
builder.comVolume(145234.20);
builder.comData(new GregorianCalendar(2012, 8, 12, 0, 0, 0));

Candlestick candle = builder.geraCandle();
```

Os *setters* aqui possuem nomes mais curtos e expressivos. Mais ainda: utilizando o padrão de projeto **fluent interface**, podemos tornar o código acima mais conciso, sem perder a legibilidade:

```
Candlestick candle = new CandleBuilder().comAbertura(40.5)
    .comFechamento(42.3).comMinimo(39.8).comMaximo(45.0)
    .comVolume(145234.20).comData(
```

```
    new GregorianCalendar(2008, 8, 12, 0, 0, 0)).geraCandle();
```

Para isso, a classe `CandleBuilder` deve usar o seguinte idiomismo:

```
public class CandleBuilder {

  private double abertura;
  // outros 5 atributos

  public CandleBuilder comAbertura(double abertura) {
    this.abertura = abertura;
    return this;
  }

  // outros 5 setters que retornam this

  public Candlestick geraCandle() {
    return new Candlestick(abertura, fechamento, minimo, maximo,
            volume, data);
  }
}
```

Escreva um código com `main` que teste essa sua nova classe. Repare como o builder parece bastante com a `StringBuilder`, que é uma classe builder que ajuda a construir `Strings` através de *fluent interface* e métodos auxiliares.

Usos famosos de Fluent Interface e DSLs

Fluent interfaces são muito usadas no Hibernate, por exemplo. O jQuery, uma famosa biblioteca de efeitos javascript, popularizou-se por causa de sua *fluent interface*. O JodaTime e o JMock são dois excelentes exemplos.

São muito usadas (e recomendadas) na construção de DSLs, Domain Specific Languages. Martin Fowler fala bastante sobre fluent interfaces nesse ótimo artigo:

http://martinfowler.com/bliki/FluentInterface.html

CAPÍTULO ANTERIOR:

Tornando-se um desenvolvedor pragmático

PRÓXIMO CAPÍTULO:

Testes Automatizados

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter