03

O desenvolver pragmático e a bolsa de valores.

O desenvolver pragmático

> **Pragmático** é aquele que se preocupa com as questões práticas, menos focado em ideologias e tentando colocar a teoria pra andar.

Esse curso tem como objetivo trazer uma visão mais prática do desenvolvimento Java através de uma experiência rica em código, onde exercitaremos diversas APIs e recursos do Java. Vale salientar que as bibliotecas em si não são os pontos mais importantes do aprendizado neste momento, mas sim as boas práticas, a cultura e um olhar mais amplo sobre o design da sua aplicação.

Aqui aprenderemos alguns **design patterns**, boas práticas de programação, a refatoração de código, preocupação com baixo acoplamento e os testes de unidades(também conhecidos como testes unitários), tudo isso passado com afinco.

A bolsa de valores

Poucas atividades humanas exercem tanto fascínio quanto o mercados de ações, assunto este que já foi abordado exaustivamente em diversos filmes, livros e em toda a cultura contemporânea. Todos nós já ouvimos nos diversos noticiários sobre os índices da Bovespa, ou sobre ações de grandes empresas que aumentaram ou diminuíram de valor.

Neste curso escolhemos o escopo da bolsa de valores por ser algo mais completo, algo que nos permitirá praticar as diversas técnicas, habilidades e boas práticas de um desenvolvedor em assunto que além de interessante, é também complexo o suficiente para expor os desafios reais do dia a dia.

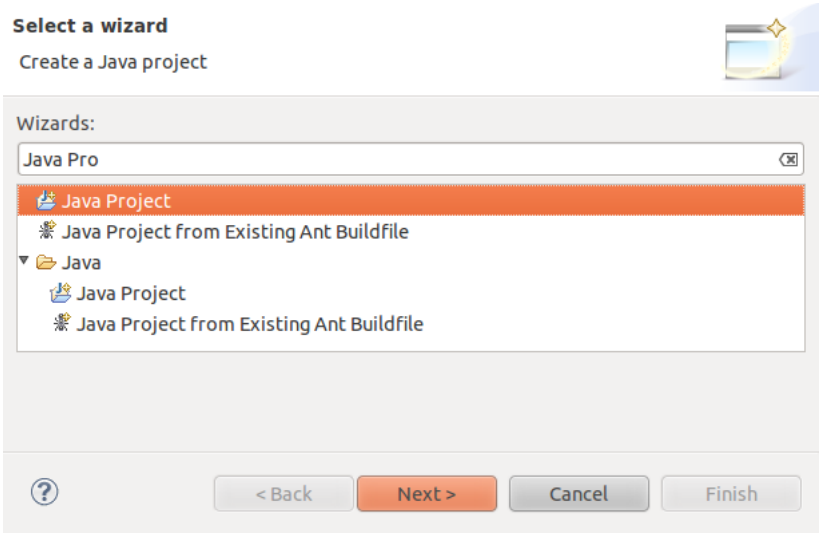
O projeto Argentum

Neste curso nós iremos desenvolver um software para análise técnica grafista de dados da bolsa de valores. Se você não sabe o que é uma análise técnica, não se preocupe, veremos isto com mais detalhes em capítulos posteriores. Por enquanto basta saber que vamos gerar gráficos que permitem a um operador da bolsa visualizar as movimentações do mercado de ações e tomar decisões baseadas nas informações que o gráfico nos fornece. Nossa aplicação irá interpretar dados sobre a bolsa de um XML, tratar e modelar eles em Java, para em seguida mostrar os gráficos pertinentes.

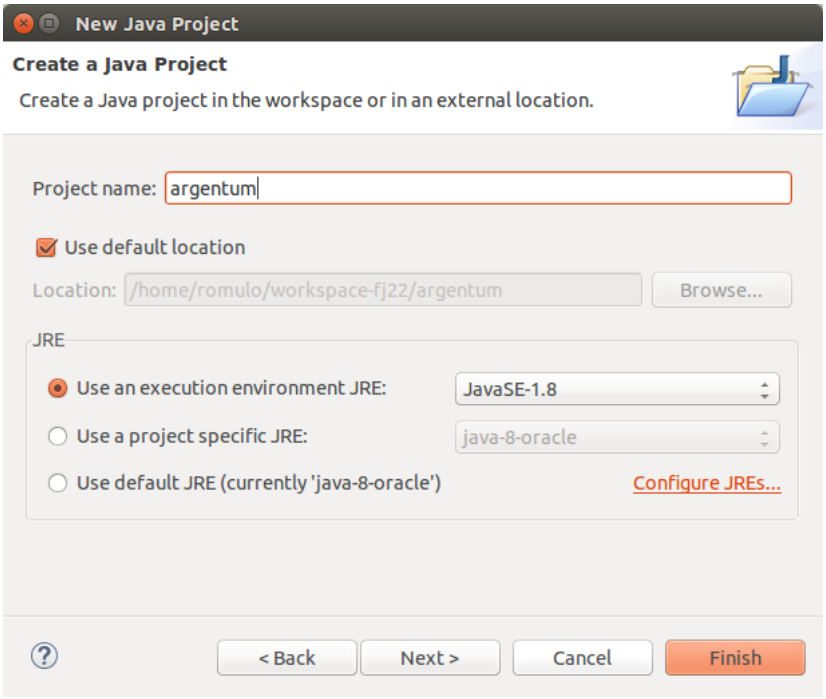
Nós escolhemos o nome **Argentum** , pois o mesmo vem do Latim que significa prata ou dinheiro.

Criando o projeto

Bom, para iniciar nosso projeto vamos abrir o Eclipse, que será a IDE utilizada neste curso. Para criar um novo projeto, com o Eclipse aberto, basta utilizar o atalho **CTRL + N**, que cria algo novo... E começar a digitar *Java Project*:

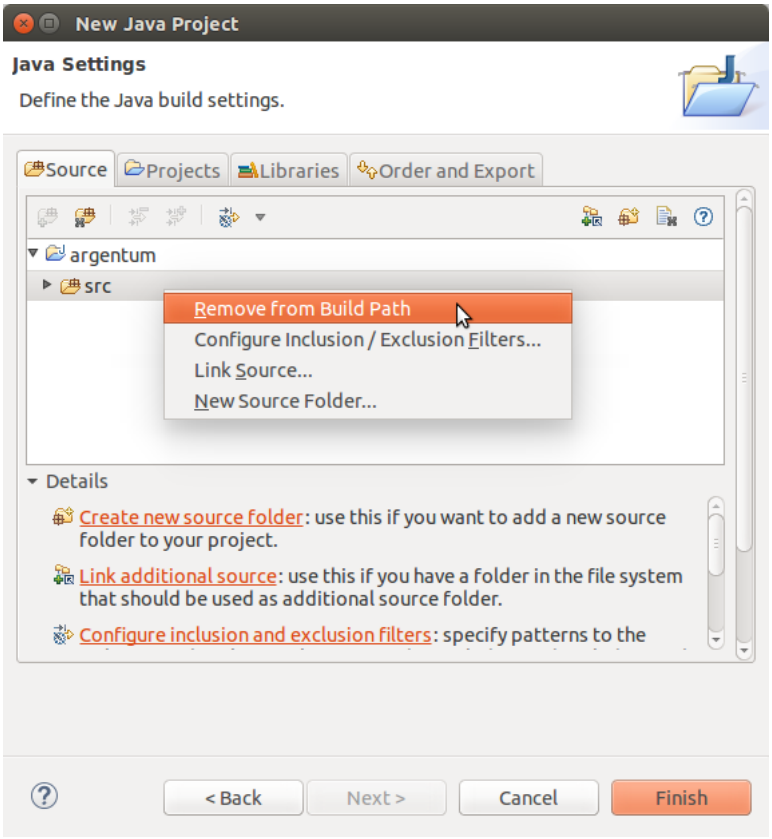


Agora, chame o projeto de **argentum** , e clique em *Next*:

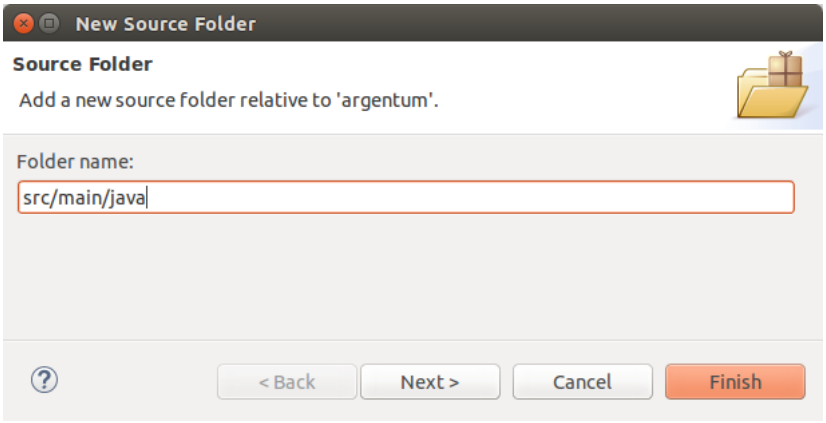


Na próxima tela, vamos definir algumas configurações do nosso projeto. Queremos mudar o diretório que conterá nosso código fonte. Faremos isso para organizar melhor nosso projeto e utilizar convenções amplamente utilizadas no mercado.

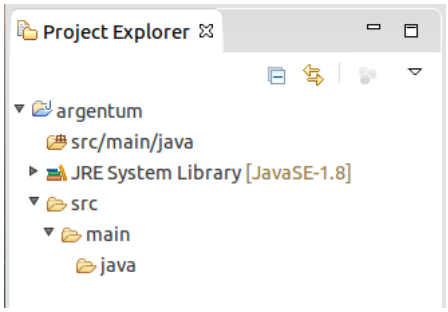
Nessa tela, **remova** o diretório `src` da lista de diretórios fontes, clicando com o botão direito e selecionando *Remove from Build Path*:



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



Para finalizar, basta clicar em *Finish*. A estrutura do seu projeto deve estar parecida com isso:



Negociação: A base do nosso projeto.

O valor de uma ação na bolsa de valores é definido, dentre de vários outros fatores, pelas operações de compra e venda dos papéis desta ação entre os operadores da bolsa. A essas operações de compra e venda nós damos o nome de **negociações**. Como é esperado, numa negociação nós temos o **preço** em que a venda (ou a compra) de uma ação foi

realizada, a **quantidade** de papéis da ação que foi transacionada e a **data** da transação. Estes 3 **atributos** farão parte do modelo da nossa classe **Negociacao**, mas vamos discutir cada um deles com mais detalhes.

Por enquanto vamos criar a nossa classe **Negociacao**, no pacote `br.com.alura.argentum.modelo` :

```
public class Negociacao {

    private ?? preco;
    private ?? quantidade;
    private ?? data;
}
```

Como guardar o preço? Aprendendo a trabalhar com dinheiro

Até agora, não paramos para pensar nos tipos das nossas variáveis, mas já é o costume de automaticamente atribuir valores à variáveis `double` , como o **preço**. 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);
```

Qual será o resultado deste código?

```
// 33.333?
// 33.333333?
// 33.3?
```

Como estamos modelando um sistema em que o escopo é a bolsa de valores, aonde acontecem um número gigantesco de negociações todos os dias, qualquer arrendondamento, por minúsculo que seja, acaba fazendo uma diferença e gerando erros. Além do que, quando estamos trabalhando com o dinheiro das outras pessoas, qualquer 1 centavo que suma ou vá para pessoa errada, pode ser considerado crime. Por isso que até em decisões simples, como a escolha do tipo de variável que guardará o preço, nós como desenvolvedores temos de ser cuidadosos, já que isso também é uma grande responsabilidade.

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

Leia mais sobre ela na própria documentação do Java (https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html).

É interessante notar que apesar de ganharmos métodos poderosos para lidar com a questão do arredondamento, a classe `BigDecimal` também vem com uma complexidade agregada, já que ao usar `BigDecimal` para representar nosso preço, todos as iterações com o preço teriam que usar os diversos métodos dela.

Porém, como o nosso intuito do curso não é focar em detalhes específicos da classe `BigDecimal`, vamos utilizar `double` na nossa implementação do sistema, para facilitar a manipulação dos dados e evitar de ter que chamar métodos diferentes toda vez que quisermos adicionar, subtrair ou alterar valores.

Sabendo disso, agora vamos atualizar a classe `Negociacao`:

```
public class Negociacao {

    private double preco;
    private ?? quantidade;
    private ?? data;
}
```

Representando a quantidade

Para representar a quantidade de ações compradas, é justo usarmos um `int`, já que a quantidade é um número inteiro de ações adquiridas. Então colocamos o tipo do atributo na nossa classe:

```
public class Negociacao {

    private double preco;
    private int quantidade;
    private ?? data;
}
```

Guardando datas da maneira correta: A API do Java 8

Manipular datas no Java sempre foi algo trabalhoso. No Java 1.0 havia apenas a classe Date, que era complicada de usar e não funcionava bem com internacionalização. Com o lançamento do Java 1.1, surgiu a classe abstrata Calendar, com muito mais recursos, porém com mutabilidade e decisões de design questionáveis.

Já agora com o Java 8, finalmente temos uma nova API do pacote `java.time`, que veio para padronizar e simplificar a maneira como lidamos com datas. E como queremos representar uma data, vamos utilizar a classe `LocalDateTime`.

Aplicando o `LocalDateTime` como o tipo da nossa data, nossa classe `Negociacao` fica assim:

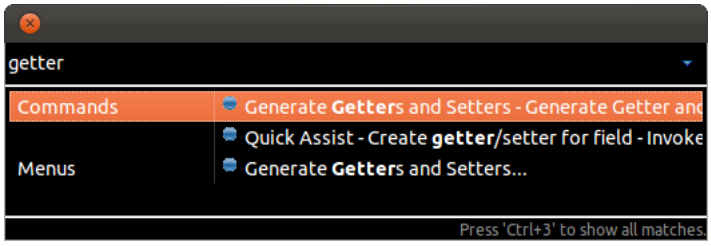
```
public class Negociacao {

    private double preco;
    private int quantidade;
    private LocalDateTime data;
}
```

Pronto, agora com os nossos tipos definidos, podemos partir para outra análise do nosso modelo `Negociacao`.

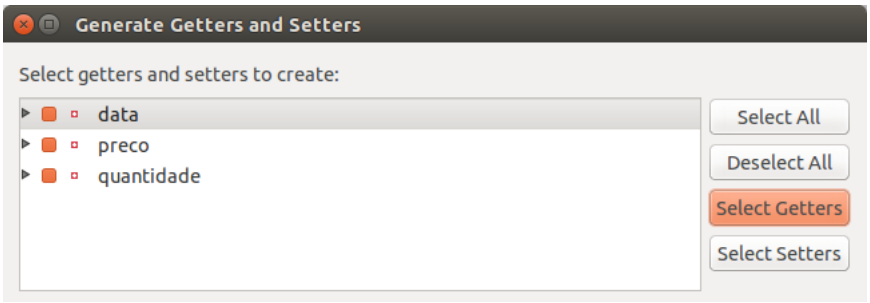
Devemos poder modificar nosso dados?

Como queremos que apenas a própria negociação tenha acesso ao conteúdo dos seus atributos, deixamo-os como `private` . Porém, se os atributos são `private` s, como acessar os seus dados? A resposta que mais vem na ponta da língua é utilizar *getters* e *setters*. Mas faça essa análise com cuidado, afinal será que é legal ter *setters* em uma negociação ? Será que deve ser possível alterar a data ou o preço de uma negociação depois dela ser criada? Quando realizamos uma compra, podemos alterar o valor da nota fiscal depois? Não é correto isso, não é mesmo?

Uma Negociação deve ser **imutável**, ou seja, depois de criada não devemos poder alterar seus dados através de *setters*, por isso vamos incluir **apenas** *getters* na nossa classe. Existe um atalho do Eclipse que nos permite criar *getters* e *setters* mais facilmente, para acessá-lo basta apertar **CTRL + 3** e começar a digitar *GGAS*, selecionando a opção *Generate Getters And Setters*:



Clique no botão *Select Getters* e confirme:

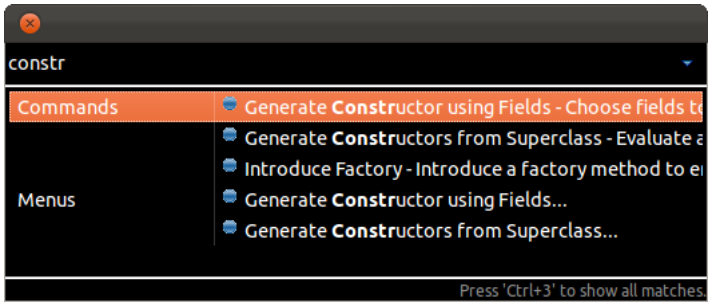



```
public class Negociacao {

    private double preco;
    private int quantidade;
    private LocalDateTime data;

    public double getPreco() {
        return preco;
    }

    public int getQuantidade() {
        return quantidade;
    }

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

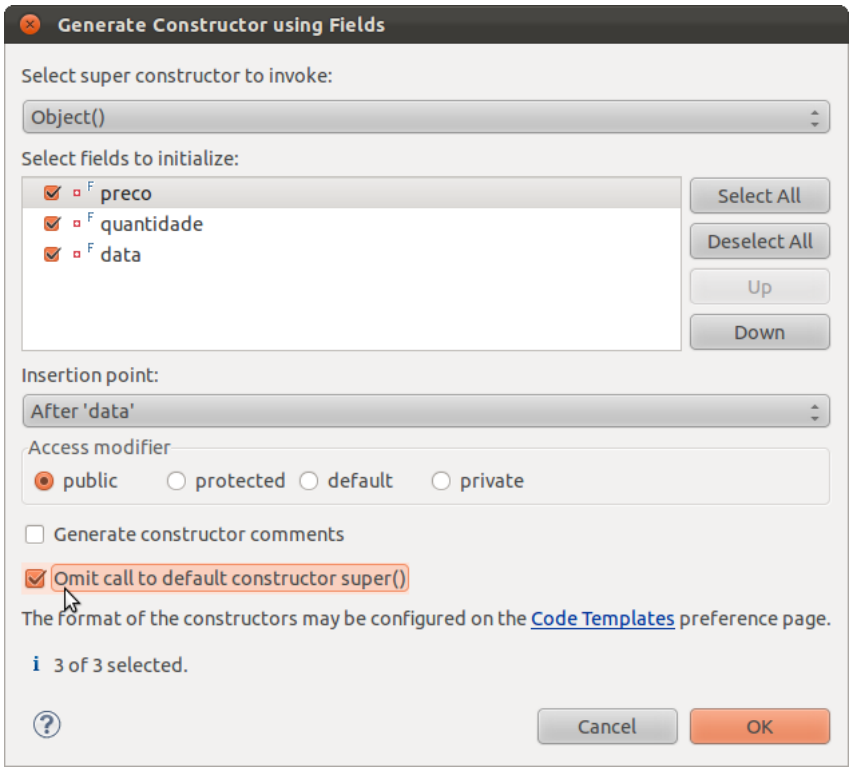
Mas agora que não incluímos os *setters*, como nós colocaremos os dados na nossa classe pela primeira vez? Simples. Através do construtor dela!

Para criar um construtor, em vez de fazermos na mão, também podemos utilizar outro atalho do eclipse com o **CTRL + 3** , e começar a digitar *contructor*. O Eclipse mostrará uma lista de opções que contêm a palavra *constructor*, basta

selecionar *Generate constructor using fields*:



Agora selecione todos os campos e marque para omitir a invocação do super, como na tela abaixo:



Pronto! Agora mande gerar. O seguinte código será gerado:

```
public class Negociacao {

    private double preco;
    private int quantidade;
    private LocalDateTime data;

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

    public double getPreco() {
        return preco;
    }

    public int getQuantidade() {
        return quantidade;
    }
```

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

Aprender estes atalhos do Eclipse facilitam bastante a vida do desenvolvedor e nos deixa muito mais agéis na hora de programar :)

Imutável de verdade

Bom, agora que eliminamos os *setters*, os dados da nossa classe não podem ser mais alterados, não é mesmo? Bem, quase isso... A nossa classe `Negociacao` ainda não é imutável de verdade. Imagine o seguinte caso: se no nosso projeto existirem outros desenvolvedores, e algum deles desavisadamente cria um método que altera o nosso preço. Todo nosso cuidado em manter a negociação imutável irá por água abaixo.

Para evitar isso, devemos usar uma palavra especial nos nosso atributos: o modificador `final`. Quando marcamos nossos atributos com `final` estamos dizendo que uma vez atribuído um valor, ele nunca mais poderá ser alterado. Isso evita que outro desenvolvedor desatento consiga alterar o valor dos nossos atributos através de um método que ele crie.

Vamos alterar nossa classe:

```
public class Negociacao {

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

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

    public double getPreco() {
        return preco;
    }

    public int getQuantidade() {
        return quantidade;
    }

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

Ficou faltando só mais um detalhe para nossa classe ficar completamente imutável. Precisamos também marcar a classe `Negociacao` como `final`, assim impedimos que outra classe herde dela. Isso impossibilita que seus métodos sejam sobrescritos ou que adicionem um novo método, garantindo assim que nossa classe é realmente imutável e que o comportamento e cateterísticas que definimos não sejam alterados de maneira alguma.

Então alterando nossa classe `Negociacao`:

```
public final class Negociacao {

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

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

    public double getPreco() {
        return preco;
    }

    public int getQuantidade() {
        return quantidade;
    }

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

Pronto! Agora temos uma classe realmente **imutável**.

O que aprendemos neste capítulo:

- O que é ser um desenvolvedor pragmático.
- O objetivo do nosso sistema **Argentum**.
- Como criar um projeto Java no Eclipse de acordo com as convenções de mercado.
- A modelar a nossa classe `Negociacao`.
- O problema de representar dinheiro como `double`.
- Como representar data de acordo com a nova API do Java 8.
- O atalho **CTRL + 3** do Eclipse para gerar *getters*, *setters* e até o construtor.
- Como deixar nossos atributos imutáveis com o modificador `final`.
- Como garantir a imutabilidade de uma classe.