

Programação Orientada aos Objetos

LCC Universidade do Minho

Ano Letivo 2015/2016

ImOObiliária — Trabalho prático

Grupo 5 62134 LCC Sérgio Oliveira
62156 LCC Carlos Silva

19 de Maio de 2016

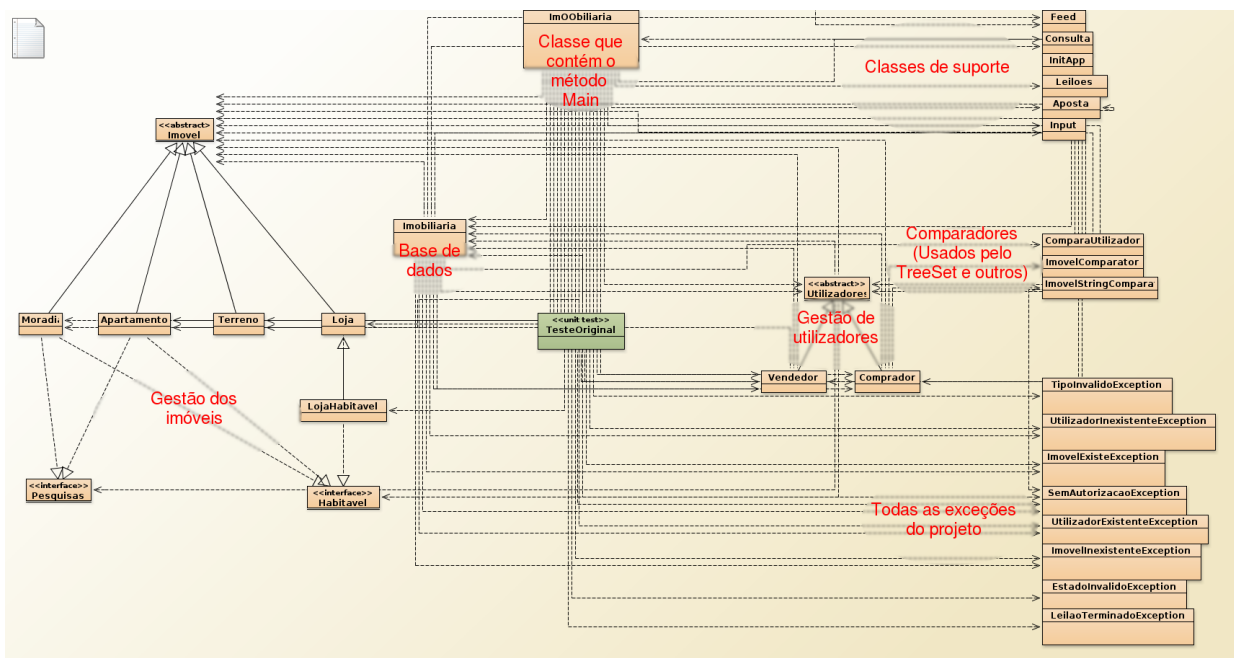


1 Introdução

Nós somos o grupo 18 da cadeira de Programação Orientada aos Objetos e pertencemos ao curso de Ciências da Computação.

Temos como objetivo construir uma interface (gráfica ou terminal, a escolha é nossa) de gestão de uma imobiliária.

Este projeto é escrito usando conhecimentos adquiridos nas aulas de Programação Orientada aos Objetos e conhecimentos pessoais. A linguagem de programação usada foi a linguagem JAVA, uma das principais linguagens orientadas aos objetos que é conhecida por ser extremamente rica em documentação online, muito dinâmica e fácil de compreender.



2 Primeiras classes

JAVA API

Vamos começar com a parte mais óbvia de qualquer projeto. Para conseguirmos compilar este programa, recorreremos a algumas bibliotecas da Interface de programação de aplicações do JAVA.

Essas bibliotecas são as seguintes:

```
java.util;  
java.io;  
java.lang;
```

Fizemos grande parte deste projeto no programa BlueJ mas devido a algumas limitações deste programa, começamos a tomar outros rumos.

```
642  
643 public static void menuPesquisar(){  
644     System.out.println("-----| |-----");  
645     System.out.println(" /      | |      \\  
646     System.out.println("|      | |      |\  
647     System.out.println("/      | |      \\  
648     System.out.println("+----- Opções -----+");  
649     System.out.println("| 1: Pesquisa por chave (morada) |");  
650     System.out.println("| 2: Pesquisa por preço mínimo |");  
651     System.out.println("| 3: Pesquisa por preço máximo |");  
652     System.out.println("| 4: Pesquisa por tipo imóvel |");  
653     System.out.println("| 5: Pesquisa por Nº de quartos |");  
654     System.out.println("| 6: Pesquisa por Nº de WC's |");  
655     System.out.println("| 7: Pesquisa por imóveis habitáveis |");  
656     System.out.println("| 8: Pesquisa por preço de imóveis habitáveis |");  
657     System.out.println("| 0: Sair da pesquisa |");  
658     System.out.println("+-----+\\n***Introduza a sua opcao: ");  
659 }
```

A interface situada na nossa classe principal com a main foi criada em Sublime Text (com a ajuda dos cursores múltiplos) em conjunto com o BlueJ e foi também utilizado ocasionalmente o IntelliJ IDEA.

Também recorreremos à "Classe que abstrai a utilização da classe Scanner", criada pelo professor F. Mário Martins e a uma espécie de Feed de notícias com recurso a uma Queue que foi tirado de um tutorial.

Sem mais demoras, passaremos a uma das principais partes do nosso projeto, os imóveis.

Imóvel

"Imóvel" é a primeira classe que começamos a fazer e é uma das duas classes abstratas do nosso programa, para conseguirmos criar imóveis mais específicos, neste caso, para dividi-los em diversas categorias para determinarmos variáveis extra.

A classe é composta pelas variáveis estado, tipoImóvel, morada, precoPedido, precoMinimo e visitas.

O clone do imóvel é ligeiramente diferente dos clones de classes normais, sendo ele:

```
abstract Imovel clone();  
...
```

Também temos dois tipos de toString() diferentes (um para os vendedores e outro para os compradores, que mais tarde falaremos deles), para que possámos ocultar a informação do preço mínimo ao comprador.

```
public String toString(){  
...  
}
```

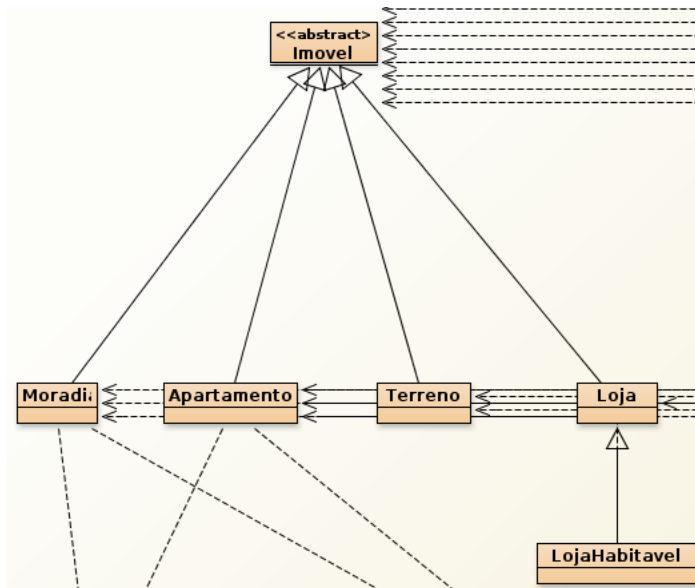
Também existe a opção de retornar, gravar e incrementar as visitas aos nossos imóveis para uso na classe Consulta, para que quando um utilizador qualquer aceder a um dado imóvel, o sistema consiga registar esse acesso.

Imovel está também povoado pelo típico equals() muito predefinido.

```
...  
public boolean equals(Object o){  
...  
}  
...
```

Como já está praticamente tudo falado da nossa classe abstrata, iremos então descer na nossa herança para elementos menos abstratos.

"Subclasses de Imóvel"



Começamos com a subclasse Moradia. Esta foi a primeira subclasse que fizemos do abstract Imóvel. Como é habitual nas subclasses, fazemos construtores importando sempre as variáveis importantes da nossa classe abstrata através do método super(), subindo na herança para a nossa superclasse.

A subclasse Apartamento traz um tipo de variável um pouco habitual nas subclasses que fizemos que é neste caso o boolean. Esta subclasse também será importante para outra classe que falaremos daqui a pouco.

Tanto a classe Moradia como a classe Apartamento têm uma String em comum (Tipo). No entanto, os tipos das duas classes diferem uma da outra. Enquanto que a classe Moradia só aceita como tipos: Isolada, Germinada, Banda e Gaveto

```
if (!tipo.equals("Isolada")&&!tipo.equals("Germinada")&&
!tipo.equals("Banda")&&!tipo.equals("Gaveto"))
throw new TipoInvalidoException("Tipo de imóvel inválido");
```

A subclasse Apartamento só aceita estes seguintes tipos: Simples, Duplex, Triplex

```
if (!tipo.equals("Simples")&&!tipo.equals("Duplex")&&!tipo.equals("Triplex"))
throw new TipoInvalidoException("Tipo de imóvel inválido");
```

Nota: Embora fora das classes, se estas condições não forem aceites, elas irão gerar erros que mais tarde iremos falar.

A subclasse Terreno segue o mesmo princípio das de cima, sendo o nome da String ligeiramente diferente, `apropriadoPara`, tal como a subclasse Loja que tem `tiponegocio` como nome da String.

Aqui entra a importância da Apartamento, pois uma classe Loja pode ser habitável. Ora, podemos fazer uma subclasse de Loja, ficando assim com uma subclasse LojaHabitavel, em que o principal objetivo é ligar dados do apartamento a uma dada loja.

Todas estas subclasses tem métodos em comum. Para facilitar a vida e para organização de código, nós criámos duas interfaces (`Pesquisa` e `Habitável`).

Pesquisas irá dar jeito para pesquisar o número de quartos ou o número de WC's de uma dada Moradia ou de um dado Apartamento.

Habitável por outro lado dá-nos jeito para saber se a nossa loja será habitável ou não e também retornar o preço pedido de um dado Imóvel (sendo Terreno a única exceção, a classe não tem a variável `preçoPedido`).

Utilizadores

Tivemos que implementar 3 outras classes para criar um cenário de gestão de utilizadores.

Temos uma classe abstrata chamada Utilizador que tem outras duas subclasses ligadas a ela, sendo elas Vendedor e Comprador.

No projeto ImOObiliaria, temos 3 estados que um utilizador pode usar:

- Utilizador não registado
- Vendedor registado
- Comprador registado

O facto de um utilizador estar registado ou não depende de um simples boolean.

Esse boolean diz-nos se o utilizador em questão já passou pelo estado de registo ou não.

NOTA: quando construímos um utilizador (vendedor ou comprador) vazio, a flag registado começa sempre com o valor "false".

Temos metido um apontador para a nossa principal "base de dados"(se é que podemos chamar assim). É também uma declaração importante na nossa superclasse que se deixássemos de lado, provavelmente não funcionaria da maneira que queríamos (é de notar que os principais métodos de procura de imóveis estão quase todas nos utilizadores). Esse apontador é feito usando:

```
...  
    private Imobiliaria imob;  
...
```

que nos permite então aceder sem problemas aos métodos principais da nossa classe Imobiliaria.

O resto das variáveis que usamos na superclasse Utilizadores é bastante normal e o básico para o funcionamento do projeto ImOObiliaria.

As subclasses tem pouco para mostrar em código, mas são extremamente precisas para que haja correta gestão dos dados.

Vendedor

Um vendedor registado consegue ter permissões avançadas ao programa, nomeadamente aceder ao menu exclusivo dos vendedores e a algumas opções do menu dos leilões.

A classe Vendedor não contém grandes operações. Tem em vez disso dois portfólios distintos: um para armazenar os imóveis que estão em venda na base de dados e outro para registar os imóveis que já foram vendidos.

O vendedor também tem a possibilidade de remover diretamente um imóvel da base de dados, utilizando para isso o apontador para a classe Imobiliaria.

Comprador

Este estado também tem as suas únicas permissões não tão avançadas como um vendedor. Um comprador pode aceder ao menu exclusivo dos compradores, onde a criação, edição ou a eliminação de todos os imóveis não são permitidos.

Um comprador também consegue ter acesso aos leilões. No entanto dar início a um dado leilão ou encerrar um leilão não são operações permitidas.

O comprador, ao contrário do vendedor, não tem nenhum portfólio de imóveis em venda ou vendidos mas sim uma lista de imóveis da escolha do comprador que possa gostar mais.

A classe comprador é das únicas classes que recorre à ordenação de listas, recorrendo para isso às classes Comparator que estão situadas no projeto.

```
... TreeSet <Imovel> (new ImovelStringComparator ()); ...
```

Ambas as classes recorrem ao método clone() para no caso de querermos registar informação de um dado utilizador sem alterar absolutamente nada do objeto original.

3 O motor do projeto

Imobiliária (Base de dados)

Esta classe é a classe que provavelmente tem o maior número de métodos do nosso projeto (a grande responsável pela criação de dados).

As variáveis de instância desta classe são as que fazem o armazenamento de grande parte dos dados dos imóveis, utilizadores e um feed de notícias .

Quase todos os nossos getters recorrem a uma lista ou map auxiliares, de forma a que retornemos corretamente os dados enquanto que os setters são praticamente normais.

A classe faz-nos a gestão do início e do encerramento da sessão atual, não continuando se a base de dados dos utilizadores não tiver conteúdo ou se nenhuma sessão estiver ativa respetivamente.

Ao iniciarmos um programa, nós temos que criar um ponto em que haja um tipo de utilizador que não seja abstrato (pois não podemos instanciar uma classe abstrata) nem esteja registado.

Para isso recorreremos a uma maneira de simular o que queríamos, criando um método `init` que inicia um Comprador sem qualquer informação mas que esteja ligado à nossa base de dados, em que a flag registado seja `false` para não ter permissões aos nossos menus especiais.

```
public void init(){atual = new Comprador ("","","","","",this,false);}
```

Ao terminar a sessão, a classe cria um ponto de situação em que o utilizador não tem informação e não está registado no sistema (`boolean registado = false`);

```
public void fechaSessao(){... }
```

Os imóveis poderão ser registados, editados, listados ou removidos através desta classe, assim como os utilizadores também poderão ser registados e listados, mas não editados ou removidos.

O algoritmo do mapeamento de imóveis e do top de visitas dos imóveis também foi inserido nesta classe e recorre a apontadores dos dois tipos de utilizadores, para que

possámos relacionar as visitas que os compradores ou os vendedores fazem aos imóveis, ou relacionar os vendedores e os seus respetivos imóveis em venda/reservados/vendidos.

Dois dos métodos mais importantes desta classe são os que nos permite a função de carregamento ou de salvaguarda de objetos, ou seja, guardar o estado inteiro do nosso programa. Para isso usamos então as classes predefinidas de JAVA `ObjectInputStream` e `ObjectOutputStream`.

ImOObiliaria (Main)

Este é o nosso arranque, a classe que tem o controlo dos principais menus de gestão do utilizador, de pesquisa, de vendedor e comprador, do leilão e do mapeamento de imóveis mais o top das consultas de imóveis do programa.

ImOObiliaria é a grande responsável por listar dados. No entanto, `inserImovel` é dos únicos métodos nesta função que utiliza o algoritmo de criação de dados em vez de listar, sendo a implementação desta função nesta classe um pouco estranha.

Nesta classe todas as variáveis estão a static para que não seja possível instanciar as mesmas variáveis noutras classes, ou seja, só existe uma instância deste campo estático.

Esta classe está dividida em 4 partes:

- Janelas;
- Menus;
- Métodos de edição de variáveis;
- Métodos de atualização de avisos.

A classe 'ImOObiliaria' terá que ser instanciada no método `InitApp`, para que se possa carregar/gerir os objetos. Para isto, existe a interface `Serializable` para que o nosso objeto guardado pela classe 'Imobiliaria' seja sempre compatível com o nosso programa. Ao ser implementado temos que ter em conta que poderá ocorrer eventos anormais, que caso aconteça, o nosso programa irá usar as `Exceptions` (que falaremos daqui a bocado) evitando que se fechará abruptamente gerando um erro.

4 Outras classes importantes

Leilões

Leilões é a nossa classe que faz a gestão de um leilão. O vendedor tem a oportunidade de meter o seu imóvel em leilão disponível nas horas que o vendedor quiser. O vendedor tem também o poder de encerrar o leilão quando quiser através de uma opção do menu.

Um comprador pode entrar no leilão e pode escolher entre duas opções:

- Fazer uma licitação com dado um limite de euros, de X em X minutos, dados também pelo mesmo.
- Fazer uma licitação imediatamente, sem recorrer ao sistema de limite de euros e de tempo.

NOTA: Independentemente da opção que o comprador escolheu, ele não pode licitar menos que a licitação anterior.

Para que esta classe funcione perfeitamente, tivemos de implementar as seguintes partes da API do JAVA:

```
import java.util.Timer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Date;
import java.util.TimerTask;
```

Aposta

Esta classe é usada como suporte à classe Leilões para que consigámos usar um

HashMap <String,Aposta>

na parte dos leilões. Ela é constituída pela identificação do comprador em questão, da informação necessária à licitação e o imóvel que o mesmo quer ganhar.

Consulta

Esta classe é muito básica, dando-nos apenas o número de visualizações que cada imóvel da base de dados tem. Cada consulta começa sem nenhuma visualização como é obvio.

```
public Consulta(){  
    this.contagem = 0;  
}
```

Feed

Esta classe contém a nossa queue de 10 posições que vai atualizando ao longo das movimentações que fazemos aos dados.

Quando a queue chega aos 10 elementos e queremos adicionar mais um elemento de informação, a queue elimina automaticamente o último elemento que tem.

5 Quando o código rebenta...

Exceptions

Sem contar com a principal classe 'Exception', temos 8 classes que usam a forma de apanhar erros ao longo do código.

O princípio das classes é simples: imprimir as mensagens de erro ou fazer qualquer outra coisa que o programador quiser.

```
public class TipoInvalidoException extends Exception
{
    TipoInvalidoException(String message){
        super(message);
    }
}
```

Cada operação arriscada é acompanhada com um mecanismo chamado 'try'. Este mecanismo não funciona sem 'catch', que faz com que detete erros no código. Quem "dispara esse alarme" é o 'throw', que interage com o 'catch', fazendo com que o código pare de executar o código dentro do try, fazendo um "bypass" a essa secção e executando o código dentro da dada Exception.

Quando não ocorre um erro grave no código, o programa executa na mesma o código dentro do try/catch e segue normalmente.

6 Conclusão

No fim de tudo, encontrámos vários tópicos que não podíamos excluir na nossa conclusão.

As principais dificuldades que não foram superadas neste projeto:

- UTF-8 (utiliza o uso de caracteres especiais).
- Leilões (comprador apostar de x em x minutos).
- Bug estúpido do double mostrar 0.0 em vez do valor devido.
- Top X de imóveis não foi inteiramente implementado.

As dificuldades que foram superadas:

- fazer catch de uma exception no lugar errado...
- Clone() (não dava o apontador do que queríamos)
- Mapeamento de imóveis dava todos os imóveis para todos os vendedores.
- Comprar um imóvel não alterava o estado do imóvel para reservado.

Não podemos deixar de dizer que este trabalho foi obviamente produtivo e só nos trouxe vantagens para conhecer o mundo dos Exceptions, visto que começamos a "atinar" com eles. A mesma coisa aconteceu com as Interfaces, que eram demasiado "abstratas" para nós antes deste projeto.