LP II - 2017.2 - 2ª Lista de Exercícios

Em todos os exercícios o aluno deve atender aos requisitos enunciados. Métodos e variáveis auxiliares podem ser criados e usados, desde que pertinentes. O aluno deve necessariamente empregar e explorar as características de orientação a objetos do Java:

- Encapsulamento (incluindo modificadores de acesso),
- Herança (de classe e interface) e polimorfismo; Classes e métodos abstratos.
- Sobrecarga de métodos;
- Tratamento e geração de Exceções;
- Uso das classes básicas (Object, por exemplo);
- Classes / pacotes
- Uso de *generics*: não recomendado, mas será aceito - se o aluno souber o que está fazendo...

(ou seja, boas praticas de programação orientada a objetos devem ser empregadas mesmo se não foram explicitamente solicitadas)

A lista deve ser entregue, deixando os arquivos com código fonte e os executáveis (.class) na máquina virtual. O professor vai avaliar o código fonte, cumprimento dos requisitos e o programa em execução, na própria máquina.

O aluno deve criar um diretório L2, dentro do seu diretório home. Dentro de L2, criar um diretório para cada exercício com os seguintes nomes: E1, E2, E3, etc. O nome da classe com o método *main()* de cada exercício deve ser Ex1, Ex2, Ex3. Mesmo que no enunciado esteja sendo solicitado um outro nome (mude o nome solicitado por estes). A correção vai ser, o máximo possível, automatizada.

Todos os exercícios devem pertencer ao pacote default. O aluno não deve colocar as classes desenvolvidas dentro de pacotes. Isso só deve ser feito quando explicitamente solicitado no exercício.

Para não termos problemas, não use caracteres acentuados, nem no código nem nos comentários.

Para todos os exercícios devem ser tratadas as exceções numéricas, de conversão, número de argumentos, de input/output, etc., além das exceções discutidas no enunciado.

"Tratamento" de exceções no estilo abaixo não serão considerados, muito mentos o *throws* sem razão (ou melhor, para se livrar das exceções).

```
try{
    // codigo sem tratamento
}catch (Exception e)
{
    System.out.println("Erro")
}
```

Exercício 1

a) Vamos reusar a classe PessoaIMC da lista 1. Se você não fez, pode fazer agora ... acrescente os métodos *get* e *set* que você achar necessários.

b) Crie uma classe chamada *MinhaListaOrdenavel*.

1) Esta classe vai conter um objeto da classe *ArrayList*, do pacote java.util. A classe *ArrayList* implementa a interface *Collection* – isso é importante para o exercício. Este objeto vai colecionar objetos da classe *PessoaIMC, que pode ser Mulher ou Homem – ou seja não vai se criar objetos PessoaIMC, mas sim objetos das classes Homem e Mulher* e, em cima dele, vamos fazer as várias ordenações.

2) Crie métodos *add (PessoaIMC p)* e *PessoaIMC get (index i)* para adicionar e resgatar, respectivamente, objetos das classes *Homem* e/ou *Mulher* do *ArrayList* interno.

3) Crie múltiplas formas de ordenar sua lista de *PessoaIMC* (nome AZ, e Z-A, peso, altura, IMC, gênero).

Para isso vamos usar a interface *Comparator* e um truque de programação em Java: definir uma classe dentro de outra e criar uma instância dela – tudo embutido.

Para cada tipo de ordenação que queremos ter, ou seja, comparando valores inteiros ou *Strings*, temos que ter uma classe que herda da *Comparator* diferente. A que compara inteiros vai pegar um campo, digamos o *peso* ou a *altura*, de dois objetos da classe *PessoaIMC* (que podem ser objetos das classes *Homem* ou *Mulher*), subtrair um do outro e isso será uma indicação se um é menor ou maior que o outro. Se for da classe *String*, podem usar os métodos da própria classe *String* para fazer a comparação. Isso é o suficiente para se usar os métodos de ordenação...

Exemplo:

```
public Comparator pesoC = new Comparator () {
   public int compare (Object p1, Object p2){ // recebe objetos PessoaIMC como Object
      double pf1, pf2;
      pf2 = (PessoaIMC) p2.getPeso();
      pf1 = (PessoaIMC) p1.getPeso();
      return (int)Math.round (pf2 - pf1);
   }
};
```

Observe que vamos ter um objeto chamado *pesoC* (o nome seria uma contração de peso+Comparator) "dentro" do (objeto) *MinhaListaOrdenavel*. Da mesma forma vamos ter vários outros comparadores encapsulados.

- Se você for bom mesmo, ordene também por gênero (Homem / Mulher)

4) Crie um método *ordena*, que vai receber uma constante, de uma "tabela" de constantes que você vai criar dentro da classe *MinhaListaOrdenavel* e vai devolver um objeto da classe *ArrayList*, com os objetos da classe *PessoaIMC* ordenados segundo o critério da constante.

Exemplo:

```
1.Alfabetica (A-Z) – nome da pessoa
2.Alfabetica (Z-A) – nome da pessoa
3.Menor Peso - crescente
4.Maior Peso - decrescente
5.Menor Altura – crescente, do mais baixo para o mais alto
5.Menor IMC - crescente, do mais baixo para o mais alto
6.Homem / Mulher – ordenar por gênero * esse é desafio
```

Para efetivar a ordenação você vai usar o método de classe *sort* da classe *Collections* (Atenção: lá no início, dissemos que *ArrayList* implementa a interface *Collection*. Agora estamos mencionando a classe *Collections* – com "s" no final – esta classe é que oferece o método *sort* – leiam a documentação). Observe que a classe *Collections* só tem métodos de classe (*static*) para ser aplicado a objetos de coleção – mais um exemplo de utilidade dos métodos de classe.

Exemplo de como estruturar o uso do sort:

```
public ArrayList ordena (int critério) {

...switch (critério) {
      case PESO:
         Collections.sort(this.[ArrayList encapsulado] , pesoC);
         // passamos o próprio ArrayList encapusado dentro de MinhaListaOrdenavel
         // e o Comparator correspondente ao critério
     case PESO_REVERSO:
        Collections.sort(this.[ArrayList encapsulado] , pesoC.reversed());
        // observe que a única diferença é a chamada a reversed()
...
   return this.[ArrayList encapsulado];
}
```

b) O programa principal vai conter um objeto da classe *MinhaListaOrdenavel* e partir dela algumas operações serão efetuadas.

- Crie um objeto da classe *MinhaListaOrdenavel.*
- Em seguida, crie "na mão" 10 objetos da classe *PessoaIMC* (*Homem* e/ou *Mulher*) e insira no objeto *MinhaListaOrdenavel.*
- Na sequência crie um menu para o usuário imprimir a lista de produtos ou sair do programa. Se o usuário optar por listar, pergunte qual o critério e liste os produtos.

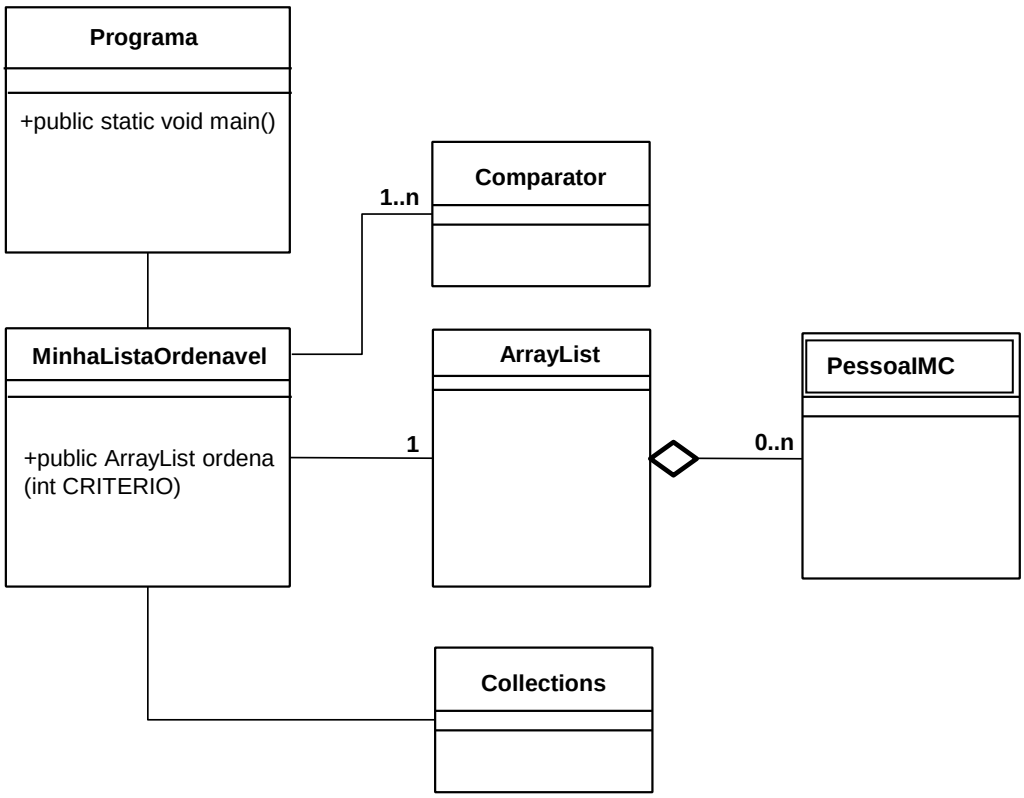

Figura 3. Diagrama de classe – Ex3

PS.: Para simplificar as coisas, pode colocar estas etapas dentro do método *main*. No próximo exercício vamos organizar melhor as coisas.

Exemplo:

```
      1.Imprimir Lista
      2.Sair
Digite sua opcao: 1
            Escolha seu modo de ordenacao
      1.Alfabetica (A-Z)
      2.Alfabetica (Z-A)
      3.Menor Peso
      4.Maior Altura
      5.Menor IMC
Digite sua opcao: 5
```

[listar as pessoas na ordem correta, exibindo as informações – isso já deveria estar incluído no *toString*]

```
      1.Imprimir Lista
      2.Sair
Digite sua opcao: 1
            Escolha seu modo de ordenacao
      1.Alfabetica (A-Z)
      2.Alfabetica (Z-A)
      3.Menor Peso
      4.Maior Altura
      5.Menor IMC
Digite sua opcao: 3
```

[listar as pessoas na ordem correta, exibindo as informações – isso já deveria estar incluído no *toString*]

Exercício 2 – Biblioteca, em 5 partes

Uma Biblioteca precisa ter controle sobre o cadastro de usuários e acervo de livros para empréstimo. Desenvolva uma aplicação que permita criar um cadastro de usuários e de livros, e acessá-los quando necessário.

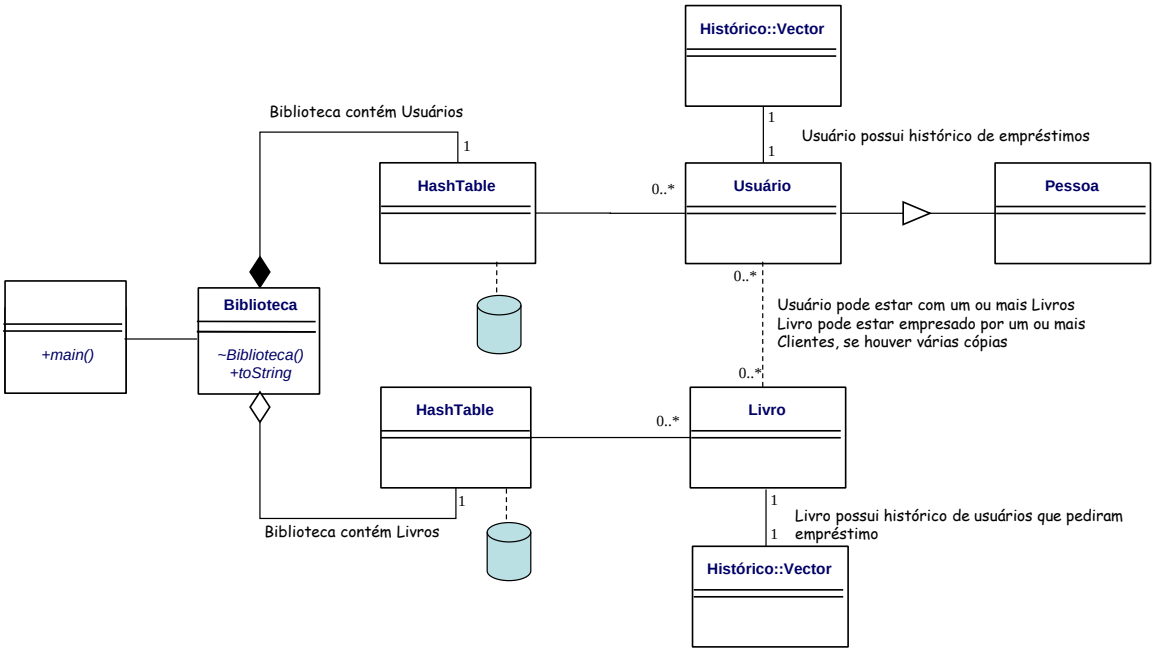

Figura 1: Modelo da Biblioteca

Requisitos:

As classes dos itens I, II, III e IV devem pertencer a um pacote *chamado lp2qXX.biblioteca* (onde XX é o número do usuário).

I) Crie uma classe *Pessoa* com os campos *nome* e *dataNasc* (da classe `GregorianCalendar(int year, int month, int dayOfMonth)`), que devem ser inicializados pelo construtor.

II) Crie uma classe *Usuário*, que estende *Pessoa*, e possua um campo *String endereço* e um campo do tipo *int códigoUsuário*. O construtor da classe *Usuário* deve inicializar todos os seus campos.

A classe *Usuário* também deve ter um campo *Histórico* da classe *ArrayList* (se já fez com *Vector*, não tem problema). Este campo deve armazenar objetos da classe *Emprestimo*.

A classe *Emprestimo* deve conter a data do empréstimo, data de devolução e o código do *Livro* solicitado para empréstimo.

A classe *Usuário* deve ter um método *addLivroHist()*, que recebe a data de locação, data de devolução e o código do *Livro* emprestado, cria um objeto *Empréstimo* com estas informações e adiciona o mesmo no *Histórico*.

III) Crie uma classe *Livro*, com campos para:

- Código do Livro (*String*)
- Título do Livro
- Categoria (que pode ser Aventura, Ficção, Romance, etc.)
- Quantidade (inteiro)
- Emprestados (inteiro)

... e os seguintes construtores:

- Um construtor que inicializa todos os seus campos
- Um construtor que recebe como parâmetro um objeto *String* e inicializa o campo do Título do Livro.

Também deve ter os seguintes métodos:

- *empresta*, que não recebe parâmetros e acerta o campo *Emprestados*. Caso todas as cópias estejam emprestadas, deve levantar a exceção confirmada *CopiaNaoDisponivelEx*;
- *devolve*, que não recebe parâmetros e acerta o campo *Emprestados*. Caso nenhuma cópia tenha sido emprestada, deve levantar a exceção *NenhumaCopiaEmprestadaEx.*

A classe *Livro* também deve ter um campo *Histórico* da classe *ArrayList*. Este campo deve armazenar objetos da classe *EmprestadoPara.*

A classe *EmprestadoPara* deve conter a data do empréstimo, data de devolução e o código *do usuário* que pegou o livro emprestado.

A classe *Livro* deve ter um método *addUsuarioHist(),* que recebe a data de locação, data de devolução e o código do *Usuário* que pediu o livro emprestado, cria um objeto *EmprestadoPara* com estas informações e adiciona o mesmo no *Histórico*.

IV) Implemente a classe *Biblioteca* com:

- um campo para o cadastro de usuários contendo um objeto da classe java.util.*Hashtable*
- um campo para o cadastro de livros contendo um objeto da classe java.util.*Hashtable.*

A classe *Biblioteca* classe deve possuir dois construtores:

- um que inicialize os campos
- um construtor que carregue o cadastro de usuários e o cadastro de livros salvos em dois arquivos distintos.

Biblioteca deve possuir os seguintes métodos:

- *cadastraUsuário*: de retorno *void*, que recebe como parâmetro um objeto da classe *Usuário* e o armazena no objeto *Hashtable correspondente*. O código do usuário deve ser utilizado como chave;
- *cadastraLivro*: de retorno *void*, recebe como parâmetro um objeto da classe *Livro* e o armazena no objeto *Hashtable correspondente*; O código do *Livro* deve ser usado como chave;
- *salvaArquivo*: de retorno *void*, recebe como parâmetros um objeto da classe *Hashtable (que pode ser o cadastro de usuários ou o acervo de livros)* e um objeto da classe *String* contendo o nome do arquivo onde o outro parâmetro será salvo;
- *lêArquivo*: de retorno *void*, recebe como parâmetros um objeto da classe *String* contendo o nome do arquivos a ser lido (como o construtor, mas que pode ser chamado a qualquer hora, e lê somente o acervo de livros ou o cadastro de usuários);
- *emprestaLivro*: recebe como parâmetros a referência a um objeto Usuário e a referência a um objeto da classe *Livro*. (as referências já devem ter sido validadas – obtidas através dos métodos *getLivro* e *getUsuário* – veja observação abaixo). Chama o método *empresta* no objeto *Livro* e atualiza o histórico no objeto *Usuário* chamando *addLivroHist.* A data do empréstimo é obtida consultando o relógio/calendário do sistema no momento da operação;
- *devolveLivro*: o mesmo definido para o método anterior, só que chama o método *devolve* no objeto *Livro*. Aqui, no entanto, se você implementou a personalização direito, pode ser verificado se o usuário está devolvendo o livro com atraso e avisar de uma multa;
- *String imprimeLivros()*: Devolve uma *string* com a lista de livros cadastrados, ordenados pelo título;
- *String imprimeUsuários()*: Devolve uma *string* com a lista de usuários cadastrados, ordenados pelo nome;
- *Livro getLivro (String cód)*: Recebe o código do livro e obtém o objeto *Livro* da *Hashtable* correspondente. Se o livro não estiver cadastrado, deve gerar a exceção *LivroNaoCadastradoEx*;
- *Usuário getUsuário (int cód)*: Recebe o código do usuário e obtém o objeto *Usuário* da Hashtable correspondente. Se o usuário não existir na *Hashtable*, deve gerar a exceção *UsuárioNaoCadastradoEx*.

Obs.: Além dos métodos listados, verifique se as classes *Pessoa*, *Usuário*, *Livro* e *Biblioteca*, devem ter, para cada campo, um método *get<NomeDoCampo>()* que retorna o conteúdo do campo.

Obs.: Para todas estas classes, exceto *Biblioteca*, sobrescrever o método *toString()* para mostrar todas as informações sobre o objeto [no caso de *Usuario* e *Livro*, deve exibir inclusive o histórico].

As classes dos itens I, II, III e IV devem pertencer a um pacote *chamado lp2qXX.biblioteca* (onde XX é o número do usuário).

V) Desenvolver o programa principal, que deve ter os seguintes módulos:

- *Manutenção*, que cria, abre e salva os arquivos (dois) que contém, cada um, uma *Hashtable* (livros e usuários);

- *Cadastro*, que cadastra usuários e livros. Deve exibir um menu com opções para: cadastrar usuários, cadastrar livros e salvar em arquivo. A opção de salvamento em arquivo deve exibir um sub-menu para que o usuário escolha se deseja salvar o cadastro de usuários ou o cadastro de livros.

- *Empréstimo*: que executa a locação de um livro para um usuário. Um menu com as opções de exibir o cadastro de livros ou fazer um empréstimo deve ser exibido. Ao fazer um empréstimo, devem ser obtidos os objetos *Livro* e *Usuário*. As exceções (livro não existe, usuário não cadastrado, cópia não disponível, etc.) devem ser tratadas. Quando um empréstimo for bem sucedido os dados devem ser escritos na tela, o registro do livro mantido (e salvo na *Hashtable* – dependendo de como for programado) e o histórico do usuário atualizado.

  Obs: no programa principal, o programador deve customizar a política da biblioteca (por isso as classes de suporte estão em um pacote e o programa principal não está). Por exemplo, pode restringir o número máximo de livros que um usuário pode emprestar de uma só vez e o número de dias que um usuário pode ficar com o livro sem pagar multa.

- *Relatório*, que permite listar o acervo de livros, o cadastro de usuários ou detalhes de um usuário ou livro específico (com seu histórico).

VI) Desenvolver dois *scripts* (um para Windows e outro para Linux) para inicializar o programa. Os *scripts* podem (devem) usar argumentos e/ou variáveis de ambiente para carregar a JVM e classes adequadamente (isso deve tornar os pacotes "independentes de localização).

Exercício 3: *Applet* – Tabela Pessoas e IMC

Você vai adaptar a 1ª questão da Lista 2 e a 3ª questão da Lista 1 – aproveitando a entrada de dados, da lista de pessoas (homens e mulheres) com ordenação.

Faça um painel para entrar com as pessoas (homens e mulheres), usando campos e botões. Mesmos campos da Lista 1. É para reusar o código.

Coloque as informações das pessoas em uma tabela gráfica (pode usar o exemplo SwitgSet2), permitindo a ordenação por qualquer campo (como feito para o Ex 1)

Faça a previsão de mensagens de erro (em janelas pop-up ou campos de texto).

Utilize componentes adicionais ou recursos que você junge necessários.

O *applet* deve ser acessado pela Internet, através da URL: 152.92.236.11/~<seu login>

Para isso você vai precisar criar um diretório *public_html*, dentro do seu diretório *home* e, lá dentro, um arquivo *index.html* com as permissões e o conteúdo adequado, além das classes em Java.

Exercício 4 (não será pontuada, mas será avaliada)

a) Crie a interface *JogoInterface*

```
public interface JogoInterface {
   public void resetar ();
   public Placar jogar ();
   public void abortar ();
}
```

b) Crie uma classe chamada *Placar.* Esta classe será usada representar o final do jogo, o resultado ou alguma outra informação que você precise passar para a estrutura de gerenciamento de jogos.

- o construtor deve receber um objeto *String*, a ser atribuído para a um campo de instância.
- um campo com que terá atribuído uma constante que indica se o jogo foi ganho ou não, com métodos *set* e *get*
- um método *toStrig()* exibindo o placar
- outros campos e métodos que você achar necessários

c) Crie uma classe chamada *MenorMaior*, que implemente *JogoInterface*, que sorteie um número aleatório **>= 1** e **<= 100**. O usuário deve tentar adivinhar qual foi o número sorteado.

O construtor deve sortear o número aleatório e guardar numa variável de instância.
O método *resetar()* pode gerar novo número aleatório
Pense no método *abortar()*
O método *jogar()* deve entrar na rotina do jogo propriamente:

Pergunta um número para o usuário, espera a sua entrada. A entrada deve ser criticada.
Ao receber o número o programa informa o número entrado tela e em seguida:

- se o usuário não acertou o número, informa se o mesmo é MAIOR ou MENOR que o número sorteado e pede uma nova entrada/tentativa de número;
- se o usuário acertar o número sorteado, o método termina e um objeto Placar deve ser criado com a informação do resultado preenchida nos seus campos. Por exemplo, uma mensagem dizendo que o usuário venceu, o numero sorteado e o número de tentativas.
- o usuário pode desistir do jogo digitando o valor 0 (zero), então o mesmo procedimento do fim de jogo deve ser usado, criando-se um objeto Placar , com uma mensagem dizendo que o usuário perdeu, o número sorteado, e o numero de tentativas, etc. e esse objeto é o retorno do método *jogar().*

Reforçando, o método *jogar()* retorna um objeto da classe Placar, com o resultado do jogo.

Exemplo:

```
O jogo Maior Menos comecou...

Digite um número: 50
O numero fornecido eh MAIOR que o sorteado.
```

```
Digite um número: 35
O numero fornecido eh MAIOR que o sorteado.
Digite um número: 25
O numero fornecido eh MAIOR que o sorteado.
Digite um número: 15
O numero fornecido eh MAIOR que o sorteado.
Digite um número: 5
O numero fornecido eh MENOR que o sorteado.
Digite um número: 9
O numero fornecido eh MENOR que o sorteado.
Digite um número: 12
O numero fornecido eh MAIOR que o sorteado.
Digite um número: 11
```

d) Crie uma classe chamada *CaraCoroa* que implemente a interface *JogoInterface*.

O jogo você conhece. Use os requisitos da classe *MaiorMenor* como base e desenvolva classe. Adapte o que for necessário. Mas, o uso da *JogoInterface* amarra propositalmente algumas coisas. Isso serve para fazer com que todas as classes de "jogos" possam ser usadas da mesma forma por uma classe "gerenciadora de jogos".

e) Crie uma classe *Ex5 (*antes o nome era *Jogo)*, classe principal do programa. A idéia é que o usuário vai executar esta classe e vai poder escolher o jogo, jogar ou abortar, jogar novamente o mesmo jogo, jogar outro jogo e terminar o programa.

Também, queremos mostrar que o uso de modularidade, interface e herança pode tornar a produção do programa mais ágil. Imagine por exemplo a criação de uma fábrica de jogos.

O método *main()* deve:
- criar uma instância de objeto da classe *Ex5* (sim!, não vamos mais usar a execução de tudo no método *main()*, estático – vamos criar um objeto mesmo)
- chamar o método *iniciar()*
- chamar o método *finalizar()*

O método *iniciar()* deve:
- perguntar ao usuário qual jogo ele deseja jogar.
- criar o objeto da classe do jogo correspondente
- chamar o método *jogar()*, passando o objeto do jogo
- fazer os laços necessários para o usuário jogar novamente
- aqui, se você for "esperto" vai preparar a estrutura de "menus" para que novos jogos sejam incluídos.

O método *jogar()* do objeto da classe *Ex5* deve:
- receber como parâmetro o objeto do jogo (ou seja, (*JogoInterface* j))
- chamar o método *jogar()* do objeto do jogo passado como parâmetro.
- com o retorno do método anterior, deve chamar o método *exibirPlacar()*, que exibe o resultado do jogo

O método *exibirPlacar(Placar p)* recebe o objeto Placar e exibe o resultado dessa "rodada"

O método *finalizar()* fecha a aplicação. Neste método, seria possível fazer uma contabilidade de quanto tempo o usuário usou o jogo, poderia totalizar alguns resultados e até pedir uma contribuição se ele gostou de usar a aplicação.

Adicione campos e métodos que você achar necessários.

Exercício 5: *Threads*- Problema do Banheiro Unissex (desafio-bônus * fale com o professor antes de fazer!)

Suponha que em um local movimentado haja apenas um banheiro, “Unissex”. O banheiro possui em seu interior cabines individuais, mas homens e mulheres não podem usá-lo ao mesmo tempo.

Vamos utilizar técnicas de programação concorrente para controlar o acesso ao banheiro e simular essa situação.

Vamos, também, criar algumas possibilidades para esta solução, usando herança.

I) Crie as classes de exceção *BanheiroOcupado* e *BanheiroVazio* que herdem de *ArrayIndexOutOfBoundsException* e exibam, respectivamente, as mensagens "Desculpe - O banheiro esta lotado" e "O banheiro esta vazio". Crie as classes *Alerta* e *Espera* que herdem de *Exception* e exibam, respectivamente, as mensagem "Homens e mulheres não podem usar o banheiro ao mesmo tempo" e “Aguarde um momento por favor”.

II) Crie a classe *Humano* com os campos *gênero* (que pode ser masculino ou feminino), *identificador* (que é um valor inteiro) e um campo contendo uma referência para um objeto da classe *Banheiro* (que será definida abaixo). O construtor da classe *Humano* recebe como parâmetros um objeto da classe *Banheiro* e um valor inteiro, e inicializa os campos correspondentes.

III) Crie a classe *Banheiro* com campos inteiros *quantidade* e *capacidade* e o array *cabines*, onde se possa alocar instâncias da classe *Humano*. Seu construtor utiliza o campo capacidade para inicializar o tamanho do array, que deve ser 5. A classe *Banheiro* deve ter os seguintes métodos:

- *void entraBanheiro(Humano h)*: Homens e mulheres não podem usar o banheiro ao mesmo tempo, logo este método deve verificar o gênero (feminino ou masculino) de quem tenta entrar no banheiro, e verificar se dentro do banheiro há pessoas do sexo oposto. Caso haja, deve lançar a exceção *Alerta*, senão, o método tenta inserir o objeto no banheiro e em caso de sucesso imprime a mensagem "<Homem ou Mulher> <identificador> Entrou". Se o banheiro estiver lotado, deve lançar a exceção *BanheiroOcupado*.

- *void saiBanheiro()*: Tenta retirar um dos objetos (Homem ou Mulher) que estão usando o banheiro. Caso o banheiro esteja vazio, lança a exceção *BanheiroVazio*.

Lembre-se de utilizar mecanismos que garantam que homens e mulheres tenham as mesmas chances de entrar no banheiro. A exceção Espera deve ser lançada nos casos em que as condições deste mecanismo não tenham sido atendidas.

IV) - Crie a classe *BanheiroExt*, que estende *Banheiro* e modifica o número de cabines disponíveis (tamanho do banheiro). Seu construtor recebe como parâmetro o novo tamanho máximo do banheiro. Os métodos *entraBanheiro* e *saiBanheiro* devem manter as mesmas funções, mas serão sobrescritos da seguinte forma:

- Terão o modificador *synchronized* acrescentado à sua assinatura, para que seja resolvido o problema da concorrência.
- Para sincronizar o acesso ao banheiro, deve ser usada a seguinte "receita":

No início do método *entraBanheiro()*:

```
while(banheiro cheio)
{
    wait(); //Espera alguém sair
}
```

No início do método *saiBanheiro()*:

```
while(banheiro vazio)
{
        wait(); //Espera alguém entrar
}
```

No fim de ambos os método:

```
notifyAll();
```

V) - Crie as Classes *Homem* e *Mulher* que estendam a classe *Humano*, implementem a interface *Runnable* e tenham o campo de classe *status* (que pode ser utilizado para garantir justiça no acesso ao banheiro). O construtor de cada classe recebe como parâmetros um objeto da classe *Banheiro* e um valor inteiro, inicializa esses campos e também o campo gênero com as palavras "Masculino" para *Homem* e "Feminino" para *Mulher*. Em cada classe implemente os métodos:

- *private void entraBanheiro()*: Tenta colocar um objeto (um homem para a classe *Homem* e uma mulher para a classe *Mulher*) no banheiro. A cada tentativa deve exibir a mensagem "Tentando entrar <Homem ou Mulher> <identificador>". A cada vez que entrar no banheiro, os homens e mulheres devem permanecer lá por um tempo aleatório (use: Thread.sleep((long)(Math.random()*100))). Se a entrada for bem-sucedida imprime a mensagem "<Homem ou Mulher> <identificador> Entrou". Caso ocorra alguma exceção do tipo *BanheiroOcupado* ou *Alerta*, deve "dormir" um tempo aleatório e tentar novamente.

- *private void saiBanheiro()*: Tenta retirar um objeto do banheiro. A cada tentativa imprime a mensagem "Saindo <homem ou mulher> <identificador>". "<Homem ou Mulher> <identificador> Saiu". Caso capture uma exceção do tipo *BanheiroVazio*, "dorme" um tempo aleatório e tenta novamente.

- *public static boolean getStatus()*: Retorna o status das instâncias da classe.

- *public static boolean setStatus()*: Altera o status das instâncias da classe.

- O método *run* deve consistir de um loop infinito, que em cada passo tenta colocar um homem ou mulher no banheiro, dorme alguns segundos, e em seguida tenta retirar alguém do banheiro.

VI) - O programa principal deve perguntar qual tipo de banheiro será criado: *Banheiro* ou *BanheiroExt* e também o número de homens e mulheres que tentarão utilizar o banheiro. O

programa inicializa o "banheiro" com o tamanho adequado (fornecido pelo usuário no caso de *BanheiroExt* e default para Banheiro) e as instâncias de *Homem* e *Mulher*, que em seguida, terão sua atividades iniciadas, como *threads*.

Exemplo de Execução:

```
1 - Banheiro  2 - Banheiro Unissex
1

Numero de Mulheres: 8
Numero de Homens: 5

Tentando Entrar - Homem 2
Homem 2 Entrou
Tentando Entrar - Homem 4
Tentando Entrar - Homem 3
Tentando Entrar - Homem 1
Homem 4 Entrou
Tentando Entrar - Mulher 1
Tentando Entrar - Homem 5
Homem 1 Entrou
Homem 3 Entrou
Tentando Entrar - Mulher 5
Mulher 1: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 5 Entrou
Mulher 5: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 3 Saiu
Homem 4 Saiu
Homem 2 Saiu
Tentando Entrar - Homem 4
Tentando Entrar - Mulher 8
Homem 1 Saiu
Homem 4: Aguarde um Momento
Mulher 8: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 5 Saiu
Tentando Entrar - Mulher 6
Tentando Entrar - Mulher 3
Tentando Entrar - Homem 4
Tentando Entrar - Mulher 8
Tentando Entrar - Mulher 5
Mulher 6 Entrou
Mulher 3 Entrou
Tentando Entrar - Mulher 2
Homem 4: Homens e mulheres nao podem usar o banheiro
Mulher 8 Entrou
Mulher 5 Entrou
Tentando Entrar - Mulher 7
Tentando Entrar - Mulher 1
Tentando Entrar - Mulher 4
Tentando Entrar – Homem 1
Mulher 2 Entrou
Mulher 7: Desculpe - O banheiro esta lotado
Mulher 1: Desculpe - O banheiro esta lotado
Mulher 4: Desculpe - O banheiro esta lotado
Homem 1: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
```