Ficha de Exercícios

Exercícios de introdução

1. Escreva uma aplicação que gere um inteiro aleatório entre 0 e 100, inclusive, e que permita ao utilizador adivinhar o número gerado, através de indicação sucessiva de valores.

2. Escreva uma aplicação que peça ao utilizador para pensar num número entre 1 e 100, inclusive. Através de várias perguntas ao utilizador, o qual responderá 'ACERTOU', 'PENSEI NUM NÚMERO MAIOR' ou 'PENSEI NUM NÚMERO MENOR', a aplicação tentará adivinhar o número. No final deve ser apresentado o número de tentativas que foram necessárias para adivinhar.

3. Escreva uma aplicação de consola que crie um *array* de inteiros, perguntando ao utilizador a dimensão do *array* e quais os valores dos seus elementos. Com base no *array* criado desenvolva funções *static* para...

a) Imprimir o *array*.
b) Calcular e imprimir o maior e o menor elemento presentes no *array*.
c) Somar todos os elementos do *array*, retornando a soma.
d) Calcular e imprimir a média dos valores do *array*.
e) Inverter todos os valores presentes no *array*, $[i_0, i_1, ..., i_{n-1}] => [i_{n-1}, ..., i_1, i_0]$

4. Defina uma classe que contenha, como membro, um *array* de inteiros com 20 elementos. Os valores a incluir neste *array* são inteiros aleatórios entre 0 e 100, inclusive. Um inteiro gerado aleatoriamente apenas deve ser colocado no *array* caso esse valor ainda não esteja armazenado, isto é, o *array* não deve conter valores repetidos. Usar uma função membro booleana (que devolve um valor booleano) para verificar se um determinado valor existe no *array*. Para cada valor gerado, deverá ser chamada a função para verificar se o mesmo já existe no *array*. Se ele existir, o valor deve ser descartado e deve ser gerado um novo valor.
A classe deve ter funções para:

a) Listar os 20 valores do *array*.
b) Mostrar a quantidade de valores que foram gerados em duplicado, ou seja, a diferença entre o total de valores gerados e o número de valores aproveitados (no caso concreto serão 20).

5. Defina uma classe, `Aposta`, que represente uma aposta no Totoloto. Uma aposta é definida por cinco números inteiros compreendidos entre 1 e 49, todos diferentes entre si e não necessariamente introduzidos de forma crescente, e um "número da sorte" compreendido entre 1 e 13 (pode ser igual a um dos 5 números indicados anteriormente). A classe quando é construída não possui qualquer número "preenchido", devendo ser disponibilizados métodos para:

a) Preencher um número de cada vez com vista à construção da aposta. O método deve possuir dois parâmetros: o primeiro é o número a associar à aposta, garantindo que não há valores repetidos, o segundo será um valor booleano para indicar se o número pertence à aposta principal ou se corresponde ao "número da sorte".

b) Verificar se a aposta está completa (cinco números e "número da sorte").
c) Preencher automaticamente uma aposta completa.
d) Comparar a aposta corrente com uma chave ganhadora. A chave sorteada é passada através de dois parâmetros: array de 5 valores inteiros e um valor adicional correspondente ao "número da sorte".
e) Fazer *reset* da aposta, eliminando todos os números eventualmente introduzidos até então.

Crie uma aplicação que utilize esta classe, permitindo a criação de instâncias da classe `Aposta` e a utilização de todas funcionalidades referidas nas alíneas anteriores.

6. Escreva uma aplicação que calcule e imprima a transposta de uma matriz. A matriz deverá ser representada através de classe adequada e deve possuir métodos adequados para as funcionalidades referidas.

7. Escreva uma aplicação que some matrizes retangulares. Uma matriz deverá ser representada através de uma classe adequada. A soma deverá ser realizada através de duas formas distintas:
a) Função membro que acumulará aos valores de uma matriz os valores de outra matriz.
b) Função estática que recebe duas matrizes e retornará uma nova matriz com o resultado da soma, ou seja, não se pretende que sejam alteradas as matrizes originais.

8. Defina uma classe que contenha, como um membro, uma matriz de *m x n* elementos. Esta classe deve ter funções para alterar os elementos da matriz, calcular a soma de cada linha, calcular a soma de cada coluna, calcular a soma de todos os seus elementos e imprimir toda a informação.
Exemplo de output para uma matriz definida programaticamente:

```
Matriz:
1    0    2    -1   3
4    3    2    1    0
1    -2   3    4    5
8    5    1    3    2

Soma das linhas: 5, 10, 11, 19
Soma das colunas: 14, 6, 8, 7, 10
Soma total: 45
```

9. Escreva uma aplicação que calcule e imprima o triângulo de Pascal. O triângulo de Pascal deverá ser representado através de uma classe própria, que deverá incluir funções para: gerar triângulo com uma determinada profundidade, gerar uma *String* representativa do triângulo (`toString()`) e mostrar o triângulo na consola. Quando o objeto for criado poderá ser indicado através do *construtor* a profundidade pretendida.

Estruturação de programas e Coleções de dados

10. Escreva uma aplicação que implemente o jogo do enforcado. Na implementação do jogo deverá garantir uma separação entre o código referente a interação com o utilizador e as classes necessárias para realizar a sua gestão. O código deverá ser estruturado nas seguintes classes:

- `JogoEnforcado`, que inclui (apenas) a função `main`;
- `JogoEnforcadoDicionario`, que permite disponibilizar as palavras a usar no jogo. Deverá possuir apenas membros estáticos para armazenar uma tabela com as palavras, e métodos para acesso à quantidade de palavras armazenadas e a uma palavra (através do seu índice);
- `JogoEnforcadoModelo`, que efetuará a gestão de todo o jogo (sem qualquer código de interação com o utilizador). Deverá conter métodos que permitem iniciar jogo (sorteando uma nova palavra), tentar/verificar uma letra, verificar fim de jogo, gerir informação de evolução de jogo: número de tentativas, letras já tentadas, ...;
- `JogoEnforcadoIU`, onde será implementada toda a interação com o utilizador.

Exemplo da classe `JogoEnforcado`:

```
public class JogoEnforcado {
    public static void main(String args[]) {
        JogoEnforcadoModelo jogo = new JogoEnforcadoModelo();
        JogoEnforcadoIU jogoIU = new JogoEnforcadoIU(jogo);
        jogoIU.jogar();
    }
}
```

11. Defina uma classe para representar a informação acerca de um relatório. Um relatório (instância da classe `Report`) é definido através dos seguintes campos:

- *Titulo* (`String`)
- *Conjunto de autores* (1ª versão: array de *strings*; 2ª versão: `ArrayList` de *strings*)
- *Texto* (`StringBuilder` ou `StringBuffer`)

A classe deve disponibilizar as seguintes funcionalidades:

- Acrescentar um autor ao relatório garantindo que não existem repetidos.
- Remover um autor, garantido a manutenção da ordem dos restantes.
- Acrescentar texto. O texto será concatenado ao final do texto já existente.
- Substituir por letras maiúsculas as primeiras letras das palavras depois de pontos finais.
- Contar as palavras do texto (as palavras podem estar separadas por mais do que um separador: espaços, tabs, mudanças de linha, vírgulas e pontos). A função deverá retornar o número de palavras.
- Contar as ocorrências de uma dada palavra. A função deverá retornar o número de ocorrências ou 0 (zero) caso a palavra não exista no texto.
- Gerar `String` com toda a informação do relatório (método `toString()`).

Implemente uma aplicação Java que permita testar todas as funcionalidades da classe `Report`.

12. Pretende-se uma aplicação para gerir os produtos de uma fábrica.

a) Os produtos, representados através de uma classe `Produto`, são identificados por um número de série (inteiro) que deve ser único e não deve poder ser modificado a partir do exterior da classe. São ainda caracterizados pela data de fabrico e também por um estado (*string*) que inicialmente possui o valor "`não testado`". Um objecto da classe `Produto` só pode ser criado com a indicação do seu número de série. A data de fabrico corresponde à data do momento de criação do objecto. Esta classe deverá ter um método booleano `testaUnidade()` que simula o controlo de qualidade. Quando esta função é invocada por um objeto, se este estiver no estado "não testado", em 90% dos casos estará OK (o estado passará de "`não testado`" para "`aprovado`"). Caso não esteja OK, o estado passará para "`reprovado`". A função `testaUnidade()` não altera o estado caso este seja "`aprovado`" ou "`reprovado`", retornando `true` se o estado é aprovado e `false` se não é. As variáveis-membro devem ser privadas, podendo ser acedidas através de funções *get* e *set*. Nesta classe deve também implementar (redefinir) as funções:
   i. `toString()` que retorna uma *string* com a descrição do objeto;
   ii. `equals()` que representa o critério de identificação de um produto (dois produtos são o mesmo se tiverem o mesmo número de série);
   iii. `hashCode()` que retorna um *hash code* adequado para o objeto.

b) Defina a classe `Fabrica` que gere um conjunto de produtos. Para além de uma coleção de produtos, a fábrica tem um nome e o número de produtos criados até ao momento (que pode ser diferente do número de elementos da coleção de produtos, uma vez que podem já ter sido alguns eliminados). Ao ser criado um objeto da classe `Fabrica` deve ser dado o seu nome, ficando, à partida, sem registo de nenhum produto. Esta classe deve ter as seguintes funções:
   i. `acrescentaProduto()` que cria o produto e acrescenta-o à fabrica;
   ii. `pesquisaProduto()` que recebe o número de série de um produto e retorna uma referência para o produto se o encontrar ou `null` se não encontrar;
   iii. `eliminaProduto()` que recebe o número de série do produto, eliminando-o se o encontrar. Retorna o valor lógico do sucesso desta operação;
   iv. `eliminaReprovados()` que elimina todos os produtos cujo estado é "`reprovado`";
   v. `testaUnidades()` que faz o controlo de qualidade a todos os produtos da fábrica;
   vi. `toString()` que retorna uma *string* com a descrição do objeto.

c) Desenvolva uma função `main` que permita testar as funcionalidades desenvolvidas nas alíneas anteriores.

Coleções de dados: List, Set e Map

13. Pretende-se uma aplicação para gerir os livros de uma biblioteca. Os livros são identificados por um código (um número inteiro positivo que representa a ordem de criação do registo dos livros na biblioteca). O registo de um livro, para além do referido código, tem obrigatoriamente informação sobre o título e os autores.

a) Defina a classe `Book` que representa este conceito de registo de um livro nesta biblioteca. Deve ser possível criar objectos da classe `Book`, dando informação sobre o título e autores, sendo, o código gerado automaticamente. As variáveis-membro devem ser privadas, podendo ser acedidas através de funções *get* e *set*. A variável-membro código não deve poder ser modificado a partir do exterior da classe. Nesta classe deve também implementar as funções:
   i. `toString()` que retorna uma *string* com a descrição do objeto;
   ii. `equals()` que representa o critério de identificação de um livro (dois livros são o mesmo se tiverem o mesmo código);
   iii. `hashCode()` que retorna o *hash code* do objeto.
   iv. *`getDummyBook()`* método estático que recebe o código de um livro e retorna uma instância da classe `Book` com o código em causa, mas sem titulo e sem autores (`null`).

b) Defina a classe `Library` que representa uma biblioteca que possui um conjunto de livros, geridos com o auxílio de um objecto `ArrayList`. Para além dos livros, a biblioteca tem um nome. Ao ser criado um objeto da classe `Library` deve ser dado o seu nome, ficando, à partida, sem registo de qualquer livro. Esta classe deve ter as seguintes funções:
   i. `addBook()` que recebe toda a informação que permite criar o registo de um livro, cria o registo e acrescenta-o à biblioteca. Deverá retornar o código do livro adicionado;
   ii. `findBook()` que recebe o código de um livro e retorna uma referência para o livro se o encontrar ou `null` se não encontrar. 1ª versão: pesquisa iterativa do livro. 2ª versão: utilização do método `indexOf()` do `ArrayList`;
   iii. `removeBook()` que recebe o código do livro, eliminando-o se o encontrar. Retorna o valor lógico do sucesso desta operação. 1ª versão: pesquisa iterativa do livro. 2ª versão: utilização do método `indexOf()` do `ArrayList`;
   iv. `toString()` que retorna uma string com a descrição do objeto.

c) Desenvolva uma função `main` que permita testar as funcionalidades desenvolvidas nas alíneas anteriores.

d) Desenvolva uma nova versão da classe `Library` recorrendo a um objeto `HashSet` para gerir o conjunto de livros.

e) Desenvolva uma nova versão da classe `Library` recorrendo a um objeto `HashMap` para gerir o conjunto de livros.

14. Pretende-se construir uma classe `Sistemas`, que oferece a seguinte funcionalidade, obtida através da utilização adequada de mapas (e eventualmente outras estruturas de dados):

```
public static void main(String[] args) {
   Sistemas d = new Sistemas();
   d.newSystem("Sistema Solar"); //nome do sistema
   d.addStar("Sistema Solar", "Sol"); // acrescenta estrela a sistema
   d.addPlanet("Sistema Solar", "Mercurio"); //acrescenta primeiro planeta
   d.addPlanet("Sistema Solar", "Venus"); //acrescenta segundo planeta
   d.addPlanet("Sistema Solar", "Terra"); //acrescenta terceiro planeta
   d.newSystem("Alfa Centauri");
   d.addStar("Alfa Centauri", "Proxima Centauri");
   d.addStar("Alfa Centauri", "Alfa Centauri A");
   d.addStar("Alfa Centauri", "Alfa Centauri B");
   d.addPlanet("Alfa Centauri", "Alfa Centauri Bb");

   System.out.println(d.getPlanet("Sistema Solar", 2)); // venus
   System.out.println(d.getStars("Alfa Centauri"));
                    // [Proxima Centauri , Alfa Centauri A, Alfa Centauri B]
   System.out.println(d.existsSystem("Xanadu")); //false
   System.out.println(d.existsSystem("Sistema Solar")); //true
}
```

15. Pretende-se construir uma classe `Dictionary`, que oferece a seguinte funcionalidade, obtida através da utilização adequada de mapas:

```
public static void main(String[] args) {
    Dictionary d = new Dictionary();

    d.add("english",   "BOOK", "book");
    d.add("francais",  "BOOK", "livre");
    d.add("portugues", "BOOK", "livro");
    d.add("english",   "YEAR", "year");
    d.add("francais",  "YEAR", "an");
    d.add("portugues", "YEAR", "ano");

    d.setLanguage("english");
    System.out.println(d.get("YEAR"));    // year
    d.setLanguage("portugues");
    System.out.println(d.get("YEAR"));    // ano
    d.setLanguage("francais");
    System.out.println(d.get("BOOK"));    // livre
}
```

16. Pretende-se construir uma classe que permite a gestão das notas de uma turma. Uma turma deve ser representada por uma classe (`Turma`) a qual deve utilizar mapas de forma adequada (sem prejuízo da utilização de outras estruturas de dados que se revelem necessárias).

```
public static void main(String[] args) {
   Turma t = new Turma();
   t.addAluno("José", 201301); //nome, nº de aluno
   t.addAluno("Luis", 201303);
   t.addAluno("Ana", 201302);

   t.addNota("Luis", 65); //nota do 1º teste do Luis
   t.addNota(201303, 80); //nota do 2º teste do Luis
   t.addNota("Luis", 85); //nota do 3º teste do Luis

   System.out.println(t.getNotaTeste(201303, 3)); //nota do 3º teste do Luis
   System.out.println(t.getNotaTeste("Luis", 2)); //nota do 2º teste do Luis
}
```

17. Pretende-se construir uma classe que permite a gestão de um inventário de produtos. A classe `Inventario` deve utilizar mapas de forma adequada para fornecer as funcionalidades descritas no seguinte exemplo (sem prejuízo da utilização de outras estruturas de dados que se revelem necessárias):

```
public static void main(String[] args) {
   Inventario t = new Inventario();
   t.addProduto("Coca", 1234, 1); //nome, código, preço
   t.addProduto("Cola", 1235, 2);
   t.addProduto("Chipi", 1236, 3);

   System.out.println(t.getPreco(1235)); //2
   System.out.println(t.getPreco("Coca"));  //1
   System.out.println(t.getCodigo("Chipi")); //1236

   System.out.println(t.getNomes()); //mostra todos os nomes:
                                     //A ORDEM NÃO É RELEVANTE!!!!
}
```

18. Pretende-se um programa que registe medições horárias de temperatura. Quando se tenta introduzir no sistema um registo de um dia com as mesmas temperaturas máximas e mínimas que outro já anteriormente introduzido, isso deve ser indicado. Verifique se o seguinte programa funciona. Caso isso não aconteça, efectue as correcções necessárias:

```
class TemperaturaDiaria {

    private int temperaturasHorarias[];
    private String responsavelMedicoes;
    private String localMedicao;

    TemperaturaDiaria(int[] temps, String resp, String local) {
        responsavelMedicoes = resp;
        localMedicao = local;
        temperaturasHorarias = new int[temps.length];
        for (int i = 0; i < temperaturasHorarias.length; i++) {
            temperaturasHorarias[i] = temps[i];
        }
    }
    public boolean equals(Object o) {
        if (!(o instanceof TemperaturaDiaria)) {
            return false;
        }
        TemperaturaDiaria outro = (TemperaturaDiaria) o;
        return getMaximo() == outro.getMaximo()
                && getMinimo() == outro.getMinimo();
    }
    public int hashCode() {
        int s = 0;
        for (int t : temperaturasHorarias) {
            s += t;
        }
        return s;
    }

    int getMaximo() {
        int max = temperaturasHorarias[0];
        for (int t : temperaturasHorarias) {
            if (t > max) {
                max = t;
            }
        }
        return max;
    }
    int getMinimo() {
        int min = temperaturasHorarias[0];
        for (int t : temperaturasHorarias) {
            if (t < min) {
                min = t;
            }
        }
        return min;
    }
}
```

```
class Temperaturas {

    private HashSet<TemperaturaDiaria> temps = new HashSet<>();

    public void acrescenta(TemperaturaDiaria td) {
        if (temps.contains(td)) {
            System.out.println("Já está registado um dia" +
                                        " com máximos e mínimos similares");
        } else {
            System.out.println("Dia com novos máximos e mínimos");
            temps.add(td);
        }
    }
}
public class Main {
   public static void main(String[] args) {
      Temperaturas temperaturas = new Temperaturas();

      int [] t_coimbra = { 1, 2, 3, 4, 5, 6, 7}; // min 1 max 7
      int [] t_lisboa  = { 1, 0, 2, 0, 2, 0, 7}; // min 0 max 7
      int [] t_porto   = { 1, 4, 2, 5, 2, 3, 7}; // min 1 max 7
      System.out.println("1 - Coimbra");
      temperaturas.acrescenta(
               new TemperaturaDiaria(t_coimbra, "Felisberto", "Coimbra") );
      System.out.println("2 - Lisboa");
      temperaturas.acrescenta(
               new TemperaturaDiaria(t_lisboa,  "Pancrácio",  "Lisboa") );
      System.out.println("3 - Porto");
      temperaturas.acrescenta(
               new TemperaturaDiaria(t_porto,   "Zeferino",   "Porto") );
   }
}
```

Herança e Polimorfismo

19. Após o desenvolvimento da aplicação do exercício 0 e depois de conhecer melhor o modo de funcionamento de uma biblioteca, verificou-se que a aplicação desenvolvida precisa de ser expandida. A biblioteca tem livros muito antigos, cuja manipulação só é permitida aos funcionários. As pessoas que frequentam a biblioteca têm acesso apenas a reproduções destas obras antigas. Também foi verificado que todos os restantes livros (mais recentes) possuem um ISBN e existe informação sobre o seu preço, pretendendo-se que ambos sejam guardados.

a) Defina uma classe `OldBook` como uma especialização de um livro genérico (`Book`) que permita definir um livro antigo. Este tipo de livro possui, adicionalmente à informação base de um livro, o número de cópias existentes (`int`).

b) Defina uma classe `RecentBook` como especialização de um livro genérico (`Book`) que permita definir um livro recente. Este tipo de livro possui, adicionalmente à informação base de um livro, o código ISBN (`String`) e o preço (`double`).

c) Garanta que o pressuposto inicial, em que dois livros são considerados iguais se possuírem o mesmo código, continua a ser cumprido (independentemente de serem livros antigos ou recentes).

d) Com a definição dos novos tipos de livro, deixou de fazer sentido serem criados livros genéricos.

   i. Altere a classe `Book` de modo que não possam ser criadas instâncias desse objeto base, mas garantindo que o normal funcionamento da aplicação.

   ii. Altere a interface com o utilizador de modo a ser possível indicar o tipo de livro, antigo ou recente, aquando da sua criação, com a indicação dos dados adicionais necessários para caracterizar cada um deles.

e) Adicione um método `toStringSorted()` à biblioteca que permita devolver informação similar ao `toString()` previamente desenvolvido, mas garantido que os livros são listados por ordem alfabética do título. Altera a interface com o utilizador para passar a usar esta nova versão.

   i. Implemente a ordenação recorrendo à interface `Comparable<T>`

   ii. Implemente a ordenação recorrendo à interface `Comparator<T>`

20. A operadora TeleAfonica oferece serviços de telecomunicações por telemóvel. No decorrer da sua atividade trabalha com conceitos tais como telemóveis, clientes, tarifas, saldos, carregamentos, chamadas, etc. A operadora permite aos seus clientes a utilização de diversos tarifários com cartões recarregáveis.

A cada cartão TeleAfonica corresponde um número de telemóvel, o saldo (`double`) e o conjunto de chamadas efectuadas. Cada chamada identifica-se pelo número do destinatário, pela duração da chamada (em segundos; `int`) e pelo seu custo (`double`).

É aos cartões que compete a funcionalidade de fazer carregamentos do saldo e de registar chamadas. Os cartões TeleAfonica aceitam sempre fazer chamadas se o saldo de que dispõem for positivo, rejeitando, se o saldo for negativo. A TeleAfonica não interrompe as chamadas, mesmo que o saldo disponível seja inferior ao custo da chamada, ficando, neste caso, com saldo negativo no final da chamada. O cartão deve também calcular o custo da chamada de acordo com o seu tarifário, registar as chamadas realizadas, e manter o saldo actualizado.

O custo da chamada é determinado pelo tarifário em vigor do cartão que efectua a chamada (não é feita distinção entre destinatários pertencentes/não-pertencentes à TeleAfonica). Os tarifários existentes na TeleAfonica são os seguintes:

- *PoucoTempo*: As chamadas têm um custo de 0,2 € por minuto.
- *Tagarela*: Este tarifário é pensado para quem gosta muito de conversar. O primeiro minuto é sempre cobrado por inteiro, a 0,4 € e os restantes a 0,02 €.

A TeleAfonica tem uma colecção de cartões que podem ser do tipo *PoucoTempo* ou *Tagarela*.
A interacção com a operadora TeleAfonica apresenta as seguintes operações:

- Acrescentar cartões lidos de ficheiro de texto
- Carregamento de um cartão, dado o número e a quantia
- Registar uma chamada, dado o número origem, destino e a duração em segundos
- Ver o saldo de um cartão, dado o número
- Imprimir factura detalhada para ficheiro de texto de um cartão, dado o número
- Listar cartões por ordem natural
- Listar cartões por ordem de números
- Listar cartões por ordem de saldos.

Desenvolva uma aplicação que permita disponibilizar as funcionalidades indicadas, recorrendo a uma hierarquia adequada de classes representativa das especificidades dos serviços referidos.

21. Pretende-se um programa em Java que permita fazer a simulação da transmissão e permanência de um vírus numa população humana. O vírus pode ser transmitido pela proximidade de uma pessoa infectada. O espaço onde vive a população em estudo é representado por numa grelha bidimensional, em que as coordenadas (abcissa e ordenada) são representadas através de valores inteiros.
O tamanho do mundo é ilimitado. Cada pessoa ocupa uma posição no mundo, definida pelas suas coordenadas que têm valores inteiros. Uma posição pode ser ocupada por mais do que uma pessoa.
O mundo é uma entidade que tem como função principal agregar as pessoas e proporcionar os mecanismos necessários à sua interacção no contexto da simulação.
Os parâmetros que descrevem o mundo são o tempo que dura a imunidade duma pessoa à doença e o tempo de duração da infecção (depois deste tempo, a pessoa morre ou fica imune).
A simulação desenrola-se através de sucessão de instantes. Em cada instante, todos os elementos que estão colocados no mundo executam uma acção específica do tipo a que pertencem. A cada novo instante de simulação o mundo manda agir todos os seus elementos e remove os elementos que morreram. Enquanto estão no sistema, as pessoas podem ser saudáveis, estar infectadas ou estar imunes ao vírus.
As pessoas deslocam-se para posições adjacentes. Uma posição adjacente a *(x, y)* é uma das seguintes posições: *(x-1, y)*, *(x+1, y)*, *(x, y-1)*, *(x, y+1)*. Uma pessoa adquire a infecção se estiver na mesma posição de outra pessoa infectada. A cada novo instante de simulação, a pessoa desloca-se uma posição, podendo ser infectada tendo em consideração o local para onde se move.
Cada pessoa pode reagir de forma diferente à infecção, depende do facto de ser jovem ou idoso (definido por uma constante no programa). Um jovem tem uma característica de resistência à infecção determinada por uma probabilidade (constante definida no programa). Se for infectado, terminado o tempo de duração da infecção, tem essa probabilidade de ficar

imune. Se não ficar imune, morre. O tempo que fica imune é um parâmetro do mundo. Durante o período de imunidade (definido através de uma constante) nada o pode infectar. Depois desse tempo, fica em estado saudável, podendo ser novamente infectado.
Se um idoso for infectado, morre depois do tempo de duração da infecção.
Defina as classes que representam o mundo, os diversos tipos de pessoas e as operações envolvidas na actividade da simulação (passagem de cada instante).

22. Uma agência de publicidade faz a divulgação de um conjunto de restaurantes, dando informação acerca de diversos tipos de menus (quais as refeições e onde são servidas). Informa, por exemplo, a um cliente de comida vegetariana, quais os restaurantes que oferecem este tipo de refeições e quais as refeições vegetarianas que constam do menu de cada um destes restaurantes.
Pretende-se uma aplicação que permita aos restaurantes registar na agência de publicidade menus de tipo vegetariano, dieta, italiano, fast-food e geral, para que constem da publicidade da referida agência.
Esta agência de publicidade tem um serviço de "Publicidade Gastronómica" representado pela classe `PublicidadeGastronomica`. Esta classe tem diversas coleções de restaurantes, uma para cada tipo de menu de que faz publicidade (colecção de restaurantes que oferecem menus do tipo vegetariano, do tipo de dieta, etc.). Nesta classe podem-se registar restaurantes com menus de tipo *vegetariano*, *dieta*, *italiano*, *fast-food* e *geral*. Para isso, basta invocar funções da classe `PublicidadeGastronomica` como `addVegetariano()`, `addDieta()`, `addItaliano()`, `addFastFood()` e `addGeral()`, respectivamente.

A função `main()` da classe que inicia a execução da aplicação cria a classe `PublicidadeGastronomica` e os restaurantes que utilizam os seus serviços publicitários.

```
public static void main(String[] args) {
   PublicidadeGastronomica publicidade = new PublicidadeGastronomica();
   new GaleriaDeOdores(publicidade);
   new UnburgerKong(publicidade);
   new Italix(publicidade);
   String [] opções = {"Vegetariano","Dieta","Italiano",
                       "Fast food", "Geral", "Sair"};
   int opcao = 0;
   //…
   while (opcao != 6) {
      //  ler opção
      switch (opcao){
         case 1: System.out.println(publicidade.divulgaVegetarianos());
            break;
         // etc...
}
```

As classes que representam os restaurantes requerem os serviços publicitários da classe `PublicidadeGastronomica` relativamente aos tipos de menus que oferecem.

O restaurante *Galeria de Odores* pretende publicitar o seguinte:

- Menu vegetariano:
  - Empadão de tofu
  - Requeijão com molho de ervas e mel
- Menu italiano:
  - Tagliateli a la carbonara
  - Risoto de legumes
  - Gelado de café
- Menu de dieta:
  - Salada mediterrânica
  - Salada niçoise
  - Maçã assada
- Menu geral:
  - Bifes de peru com molho de cenoura
  - Bacalhau com natas
  - Salmão grelhado
  - Arroz doce

O restaurante *Unburger Kong* pretende publicitar o seguinte:

- Menu fast food:
  - FishBurger
  - ChickenBurger
  - Tarte de maçã com gelado
- Menu vegetariano:
  - BeanBurger
  - TofuBurger
  - Sorvete de maçã

O restaurante *Itálix* pretende publicitar o seguinte:

- Menu italiano:
  - Lasanha de dourada com Mozarella
  - Lasanha do mar
  - Tiramisu
- Menu geral:
  - Lulas recheadas com presunto
  - Leite creme

23. Pretende-se desenvolver um programa em Java que permita fazer a gestão de uma frota de veículos de uma empresa de transportes, recorrendo a herança de classes e interfaces.
Existem vários tipos de veículo, mas todos eles possuem como características comuns uma matrícula (`String`) e um ano de construção, não existindo dois veículos com matrículas iguais.
Para gerir a frota deve ser definida uma classe `Frota` que, internamente, deve usar uma única coleção do tipo *Set* para armazenar todos veículos, independentemente do seu tipo.
A caracterização dos diferentes tipos de veículos deve ser realizada com o auxílio das seguintes interfaces:

```
interface IPassengers {
   int getNumberPassengers();
}

interface IMaxLoad {
   int getMaxWeight();
}
```

A empresa possui os seguintes tipos de veículos que deverão ter classes adequadas para os representar:

- Veículos ligeiros de passageiros, que possui um limite de passageiros;
- Veículos pesados de passageiros, que possui um limite de passageiros e de carga;
- Veículos de carga, que possui um limite de carga.

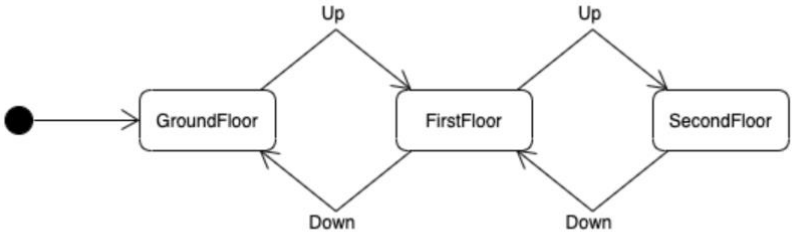
Pretende-se que a aplicação permita as seguintes funcionalidades:

- Adicionar um novo veículo à frota, perguntando qual é o tipo e questionando depois o utilizador sobre os dados específicos de cada tipo;
- Remover um veículo, indicando a matrícula;
- Listar todos os veículos;
- Listar todos os veículos que possuem um número máximo de passageiros, ordenado pelo número de passageiros;
- Listar todos os veículos que possuem um limite máximo de carga, ordenado por ordem inversa desse máximo (primeiro os que possuem um limite maior)

Desenvolva toda a aplicação, tendo o cuidado de separar as classes em packages adequados, devendo garantir, pelo menos, a separação das classes que permitem gerir o modelo de dados das classes que disponibilizam uma interface simples com o utilizador.

FSM – Finite-State Machine

24. Considere o seguinte diagrama de estados, que descreve o funcionamento de um elevador num prédio com 3 pisos.



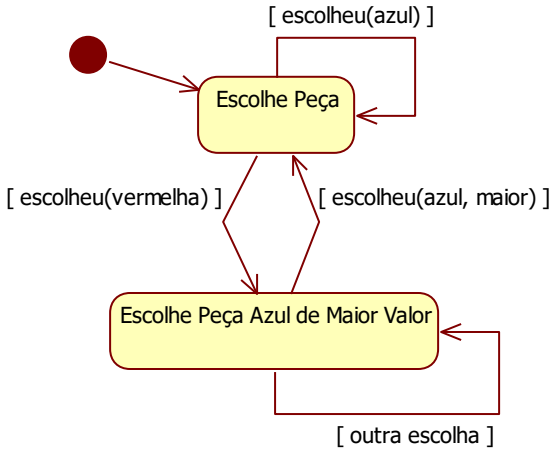
a) Construa as classes necessárias para implementar esta máquina de estados.
   i. Em cada classe que representa um estado, deve criar métodos adequados para representar as transições possíveis a partir do respetivo estado (*Up* e/ou *Down*).
   ii. Desenvolva uma classe que represente a interação com o utilizador, onde é permitido desencadear as ações (*Up* e *Down*) sobre a máquina de estados, independentemente do seu estado atual (justificação: para verificação da robustez da máquina de estados, perante ações não previstas)
   iii. Altere a interação com o utilizador, criando métodos que apresentem as opções adequadas para cada estado (i.e: possibilidade de carregar nos botões para subir ou descer no 1º piso, mas apenas carregar no botão para subir no R/C).

b) Faça as modificações necessárias para garantir a seguinte funcionalidade:
   i. Quando o utilizador carrega no botão para subir ou descer, existe uma probabilidade de o elevador ficar avariado de 10% se ocorrer no R/C, de 20% no 1º andar e de 30% no 2º andar. Sempre que é detectado um erro, o elevador entra num novo estado em que deixa de aceitar os comandos para subir e descer. O elevador só abandona este estado após a chave de segurança (*String*) ser aplicada, regressando ao estado em que se encontrava quando o erro ocorreu.

25. Considere o seguinte jogo:

"O jogador começa o jogo com 10 peças (5 vermelhas e 5 azuis). Cada uma das peças tem um número aleatório entre 1 e 4. Em cada jogada, deve largar uma das peças à sua escolha. Se largar uma peça azul, ganha esse número de pontos. Se largar uma peça vermelha, deve escolher uma peça azul de valor superior e largá-la também. Nesse caso recebe um número de peças aleatórias iguais ao número da peça vermelha que largou. O objectivo do jogo é conseguir a maior pontuação possível. O jogador pode terminar o jogo em qualquer altura."

O diagrama de estados seguinte ilustra o fluxo do jogo:



Dado que as classes seguintes são interdependentes, leia atentamente TODAS as alíneas antes de começar a resolver cada uma delas.

a) Construa as classes necessárias para representar o diagrama de estados. Deve implementar na interacção com o utilizador, correspondendo a cada um dos estados, um método para solicitar a sua opção (por exemplo, qual a peça que pretende largar).

b) Construa a classe abstracta, `Peca`, que representa uma peça com um valor entre 1 e 4. Esta peça deve ser criada com um valor aleatório na gama indicada. Deve conter métodos abstractos com protótipo:

```
abstract public Estado larga();
abstract public Estado largaSegunda(Peca pecaAnterior);
```

onde `Estado` é uma classe abstracta que representa um estado da máquina de estados indicada. O método larga é invocado quando a peça é largada (no estado *Escolhe peça*), enquanto que o método `largaSegunda()` é chamado quando a peça é escolhida para ser largada no estado *Escolhe Peça Azul de Maior Valor*.

c) Crie as classes derivadas `PecaAzul` e `PecaVermelha`.

d) Construa uma classe `Jogo` que deverá conter a pontuação do jogo, peças atualmente na mão do jogador e a máquina de estados que indica o estado do jogo. Deve conter um método `inicio()` que dá inicio a um novo jogo, bem como outros métodos auxiliares que venha a achar necessários.

26. Considere um jogo de dados com as seguintes regras:

- O jogador lança inicialmente 10 dados. Em seguida, pode opcionalmente escolher recolher e relançar simultaneamente qualquer número dos dados lançados. Este processo pode ser repetido até 3 vezes, ou seja, há no máximo 3 relançamentos.
- O jogador pode em seguida recolher qualquer número de dados lançados. O jogador ganha pontos por cada sequência de 3 ou mais números (1 ponto base +1 ponto por cada número em excesso do terceiro) ou por cada 3 ou mais números iguais (1 ponto base + 1 ponto por cada número em excesso do terceiro). O mesmo dado pode contar apenas para uma única sequência ou conjunto de números iguais. Por exemplo, caso os dados lançados sejam 2,3,3,3,4,5, o jogador pode remover a sequência (2,3,4,5) para ganhar 2 pontos ou os dados (3,3,3) para ganhar um ponto, mas não ambos.
- O jogo continua com os dados remanescentes, até que restem 3 ou menos dados. Caso restem exatamente 3 dados, o jogador recebe 1 ponto adicional antes do jogo terminar.

Considere o seguinte exemplo de utilização:

```
public class Principal {

   public static void main(String args[]) {
      JogoDados j = new OMeuJogo();
      j.comecaJogo();
      j.lancaDados();
      System.out.println(j.getDadosMesa()); // 1 3 2 6 5 2 3 1 2 2
      j.recolheDado(1);
      System.out.println(j.getDadosMesa()); // 1 2 6 5 2 3 1 2 2
      j.recolheDado(0);
      System.out.println(j.getDadosMesa()); // 2 6 5 2 3 1 2 2
      j.lancaDados();
      System.out.println(j.getDadosMesa()); // 2 6 5 2 3 1 2 2 3 4
      j.avanca();
      j.recolheDado(0);j.recolheDado(2);j.recolheDado(4); //recolhe dados com 2,2,2
      j.avanca(); //ganha 1 ponto (não recolhe o quarto dado com '2')
      System.out.println(j.getDadosMesa()); // 6 5 3 1 2 3 4
      j.recolheDado(3);j.recolheDado(3); j.recolheDado(3);j.recolheDado(3);
      j.recolheDado(1);j.recolheDado(0);
      j.avanca(); //ganha 4 pontos por sequência 1,2,3,4,5,6
      if(j.terminou()) //indica se jogo terminou
         return j.getPontuacao();  // 5
   }
}
```

27. Considere a seguinte descrição de um jogo:

Inicialmente, um saco é preenchido com 10 bolas brancas e 10 bolas pretas. As bolas vão sendo retiradas do saco de acordo com as regras que se explicam mais à frente, sendo o objetivo do jogo acumular o maior número possível de bolas brancas.

O jogo desenrola-se em duas fases distintas, que se repetem sucessivamente até que o saco esteja vazio ou o jogador pretenda terminar.

Quando o jogo se encontra na primeira fase, o jogador pode optar entre terminar o jogo ou efectuar uma aposta. Se apostar, então deve colocar de parte uma ou mais bolas brancas que tenha ganho anteriormente. Inicialmente o jogador não possui qualquer bola, pelo que deverão ser aceites apostas com 0 (zero) bolas.

Na segunda fase, depois da aposta realizada, o jogador retira aleatoriamente uma bola do saco e o resultado dependerá da cor dessa bola:

- Quando é retirada uma bola branca, esta é acrescentada à pilha de pontuação. Se o jogador fez uma aposta, recupera o valor apostado devolvendo-o para a pilha de pontuação. Em seguida retira do saco um conjunto de bolas em número igual ao da aposta realizada. As bolas pretas retiradas são eliminadas do jogo. As bolas brancas que forem retiradas neste processo são devolvidas para dentro do saco.
- Quando é retirada uma bola preta, esta é removida do jogo. Caso tenha efetuado uma aposta, ela é perdida (as bolas brancas correspondentes a essa aposta são removidas do jogo). Em seguida, são possíveis duas opções, à escolha do jogador:
    i. Remover do jogo uma bola branca da pilha de pontuação.
    ii. Tirar duas bolas aleatórias do saco, devolvendo as bolas pretas ao saco e removendo de jogo as brancas.

O jogo continua em seguida na primeira fase.

De cada vez que forem retiradas bolas o saco deve ser devidamente agitado para baralhar as bolas.

27. B. (Versão 2 do exercício 27) Considere a seguinte descrição de um jogo:

Inicialmente, um saco é preenchido com 10 bolas brancas e 10 bolas pretas. As bolas vão sendo retiradas do saco de acordo com as regras que se explicam mais à frente, sendo o objetivo do jogo acumular o maior número possível de bolas brancas.

Quando o jogo é iniciado, o jogador possui 3 bolas brancas (para além das bolas existentes no saco) na sua carteira.

O jogo desenrola-se em duas fases distintas, que se repetem sucessivamente até que o saco esteja vazio ou o jogador não possua peças brancas para apostar.

Quando o jogo se encontra na primeira fase, o jogador pode efectuar uma aposta em como vai ser tirada do saco uma bola branca. Ao apostar, coloca no tabuleiro de apostas uma ou mais bolas brancas que tenha na sua posse.

Na segunda fase, depois da aposta realizada, o jogador retira aleatoriamente uma bola do saco e o resultado dependerá da cor dessa bola:

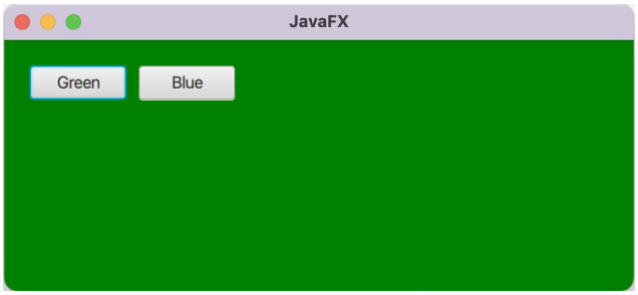
- Quando é retirada uma bola branca, esta é acrescentada à carteira. Se o jogador fez uma aposta, recupera o valor apostado (colocado no tabuleiro de apostas) devolvendo-o para a carteira. Em seguida retira do saco um conjunto de bolas em número igual ao da aposta realizada. As bolas pretas retiradas são eliminadas do jogo. As bolas brancas que forem retiradas neste processo são devolvidas para dentro do saco.

- Quando é retirada uma bola preta, esta é removida do jogo. Caso tenha efetuado uma aposta, ela é perdida (as bolas brancas correspondentes a essa aposta são removidas do jogo). Em seguida, são possíveis duas opções, à escolha do jogador:
    i. Remover do jogo uma bola branca da sua carteira.
    ii. Tirar duas bolas aleatórias do saco, devolvendo as bolas pretas ao saco e removendo de jogo as brancas.

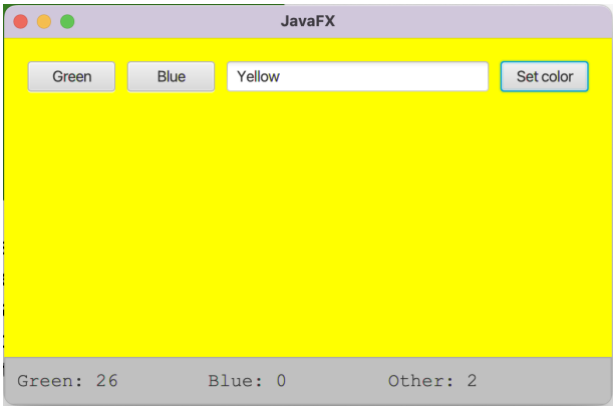
O jogo continua em seguida na primeira fase.
De cada vez que forem retiradas bolas o saco deve ser devidamente agitado para baralhar as bolas.
Quando o jogo termina deve ser mostrado o histórico de jogadas desde o início do jogo. Após a consulta do histórico, o programa deverá perguntar se o jogador pretende jogar novamente.

JavaFX

28. Escreva uma aplicação gráfica com a seguinte aparência e funcionamento:

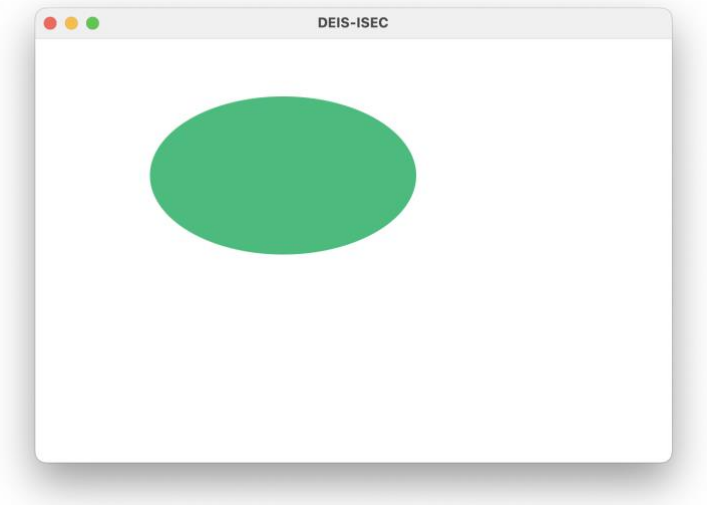


a) A aplicação deverá apresentar dois botões, "Verde" e "Azul", e, ao pressionar um botão, a cor de fundo da janela deve mudar para a cor que o botão indica.
b) Acrescentar à aplicação desenvolvida as seguintes funcionalidades:
   i. uma caixa para entrada de texto, onde o utilizador poderá escrever o nome de uma cor ou o RGB correspondente, e um botão que aplica essa cor como cor de fundo da janela. Caso a cor escrita seja inválida o fundo da janela ficará com a cor preta.
   ii. Acrescentar uma *label* ao fundo da janela que informe quantas vezes foi pressionado cada um dos botões.



29. Pretende-se desenvolver uma aplicação gráfica JavaFX que permita desenhar uma elipse cuja posição e dimensões são definidas por cliques do rato. Quando se começa a pressionar o rato define-se canto superior esquerdo do retângulo que circunscreve a elipse. Quando se deixa de pressionar o rato define-se o canto inferior direito desse retângulo. A cor da elipse é definida aleatoriamente.

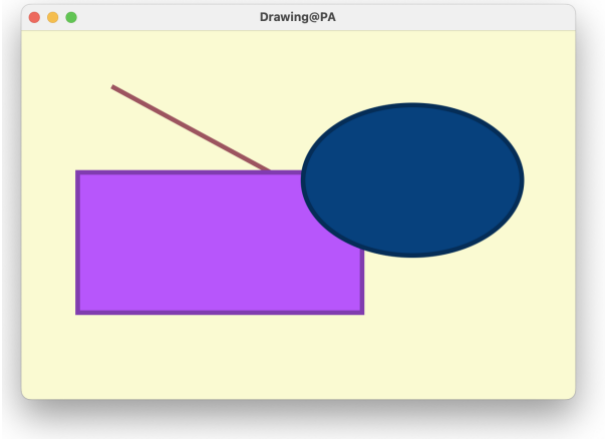
a) Desenvolva a aplicação descrita e exemplificada na imagem seguinte. Para a resolução deste exercício utilize um objeto `Ellipse` (classe derivada da classe `Shape`) colocado no contexto de um objeto `Pane`.



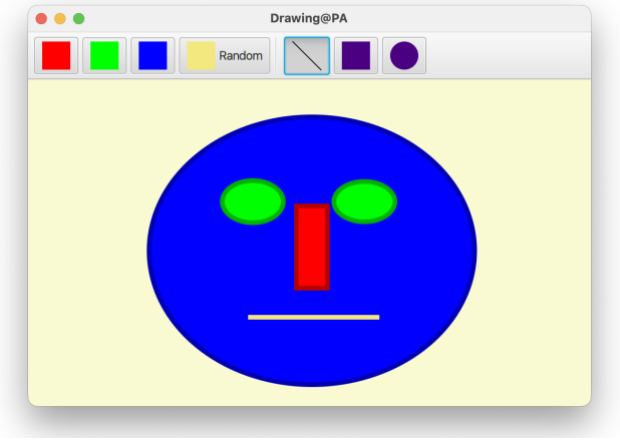
b) Adicione a seguinte funcionalidade ao programa desenvolvido: caso a tecla `Ctrl` esteja pressionada quando o primeiro ponto é pressionado então esse ponto será considerado o centro da figura e o movimento do rato definirá as dimensões da `Ellipse`.

30. Desenvolva uma aplicação *JavaFX* que permita desenhar diferentes figuras: linhas, ovais, ou retângulos, de diferentes cores. A aplicação deverá conseguir gerir e mostrar um conjunto de figuras com diferentes cores e tipos. As coordenadas da figura que está a ser desenhada são definidas por dois pontos indicados através do movimento do rato: o primeiro ponto será detetado quando o botão do rato é pressionado (sem soltar) e o segundo ponto definido pelo local onde o botão do rato deixa de ser pressionado. Entre estes dois pontos, o rato é arrastado, devendo durante esse movimento ser mostrada a figura de acordo com o ponto atual.

a) Crie todo o programa que permita apresentar as funcionalidades descritas. Nesta primeira versão quando o movimento do rato é realizado de forma simples é desenhada uma linha. Se no momento em que o botão do rato é pressionado, a tecla `Ctrl` estiver a ser também pressionada então será desenhado um retângulo. Caso seja a tecla `Alt` então deverá ser desenhada uma oval. A cor da figura deverá ser sorteada no momento em que é criada. As figuras deverão ser desenhadas recorrendo aos objetos `Canvas` e `GraphicsContext`.



b) Altere o programa de modo a que o tipo de figura e a cor possam ser selecionadas recorrendo a botões adequados numa barra de opções (*toolbar*). A cor poderá ser selecionada entre vermelho, verde, azul ou uma cor aleatória. Implemente esta nova versão recorrendo a objetos `ToolBar` e `ToggleButton`.

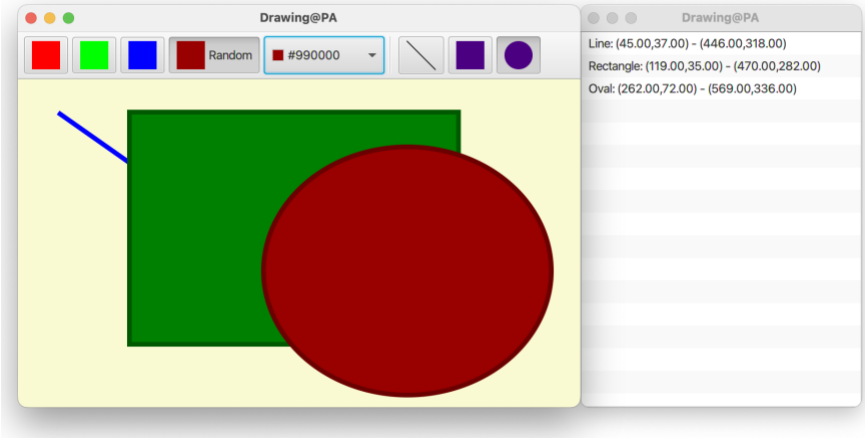


c) Adicione um objeto `ColorPicker` à `ToolBar` para poder selecionar uma cor.

d) Altere a aplicação de modo que as atualizações da interface gráfica sejam realizadas de forma "automática", recorrendo às funcionalidades proporcionadas pelas *property-change*. Adicione uma classe que faça a gestão do acesso ao modelo de dados, despoletando *property-change* adequadas para sinalizar as alterações e situações de interesse para toda a aplicação.

   i. Configure a aplicação para possuir uma segunda janela (`Stage`), similar à janela principal, para verificar o bom funcionamento das atualizações através de *property-change*.

e) Adicione uma segunda janela, relacionada com o mesmo modelo de dados, que mostre a lista de figuras usando um objeto `ListView`. Na `ListView` deverão aparecer o tipo de figura e as coordenadas correspondentes aos pontos que definem as diversas figuras.



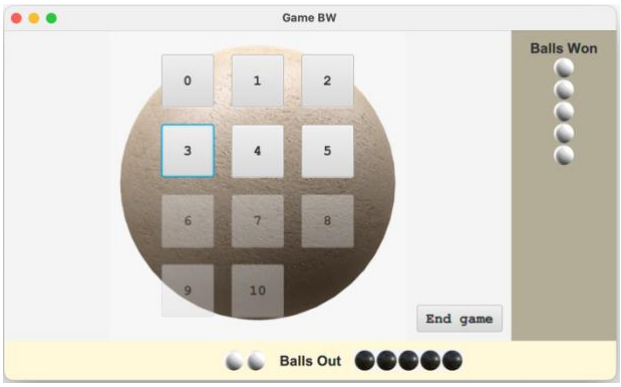
   i. Quando for realizado um *double-click* com o rato num item da lista, esse item deverá ser eliminado.

f) Adicione um *menu* à aplicação de modo a disponibilizar as seguintes funcionalidades:
   i. `File`
      - `new` – criar um novo desenho;
      - `open` – ler um desenho previamente gravado (usar `FileChooser`);
      - `save` – gravar o desenho atual (usar `FileChooser`);
      - `exit` – sair da aplicação.
   ii. `Edit`
      - `undo` – permitir a remoção da última figura introduzida;
      - `redo` – reinserir uma figura removida pela última operação de undo.

g) Adicione um *menu* de contexto à `ListView` criada anteriormente para disponibilizar as seguintes funcionalidades:
   i. `Delete` – apagar o item selecionado;
   ii. `Change` – alterar alguma informação do item selecionado (cor, tipo de figura ou outra informação à sua escolha).

31. Desenvolva uma versão do jogo do exercício 27 com interface com o utilizador em *JavaFX*.

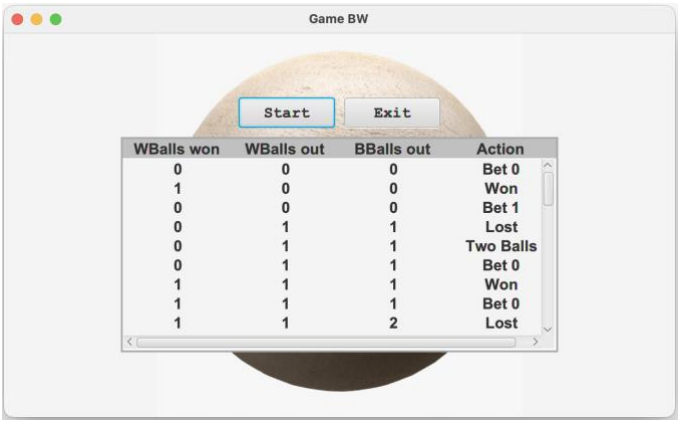
a) Tendo por base modelo de dados e máquina de estados desenvolvidos anteriormente, crie a interface referida. Adicione as funcionalidades proporcionadas pelas "*Property Change*" para atualizar a interface de acordo com a evolução da máquina de estados. Poderá definir os ecrãs de acordo com o seu gosto, apresentando-se de seguida apenas alguns *screenshots* que poderão ser seguidos como orientação para esse desenvolvimento. Deve ser garantido que as informações mostradas nos *screenshots* de exemplificação também são mostradas na interface desenvolvida.



b) Adicione um novo estado à máquina de estados que permita mostrar um resumo com os resultados finais alcançados (número de bolas brancas ganhas, número de bolas brancas de fora e número de bolas pretas de fora). Ver *screenshot* exemplificativo.

i. Altere os ecrãs respeitantes aos estados em que não se está a desenrolar o jogo de forma a não mostrar os painéis com as informações de jogo.

c) Altere a aplicação desenvolvida de modo a que no final seja mostrado um histórico das jogadas realizadas e da quantidade de bolas em cada etapa. Ver *screenshot* exemplificativo.



32. Desenvolva uma máquina de calcular básica em JavaFX, similar à mostrada na imagem. Deverá fazer a definição do ecrã recorrendo a ficheiros FXML.