Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0040 - Linguagem de Programação 2
Aula11 - Lista de exercícios

- Esta lista de exercício é composta por 3 questões. Sendo uma fácil, uma média e uma difícil.
- As questões valem, respectivamente, 20%, 30% e 50%.
- Não se assustem com o tamanho dos textos.
- Serão aceitos apenas arquivos **.zip**.
- Não será aceita nenhuma questão feita no BlueJ.
- Não serão aceitos arquivos enviados por e-mail.

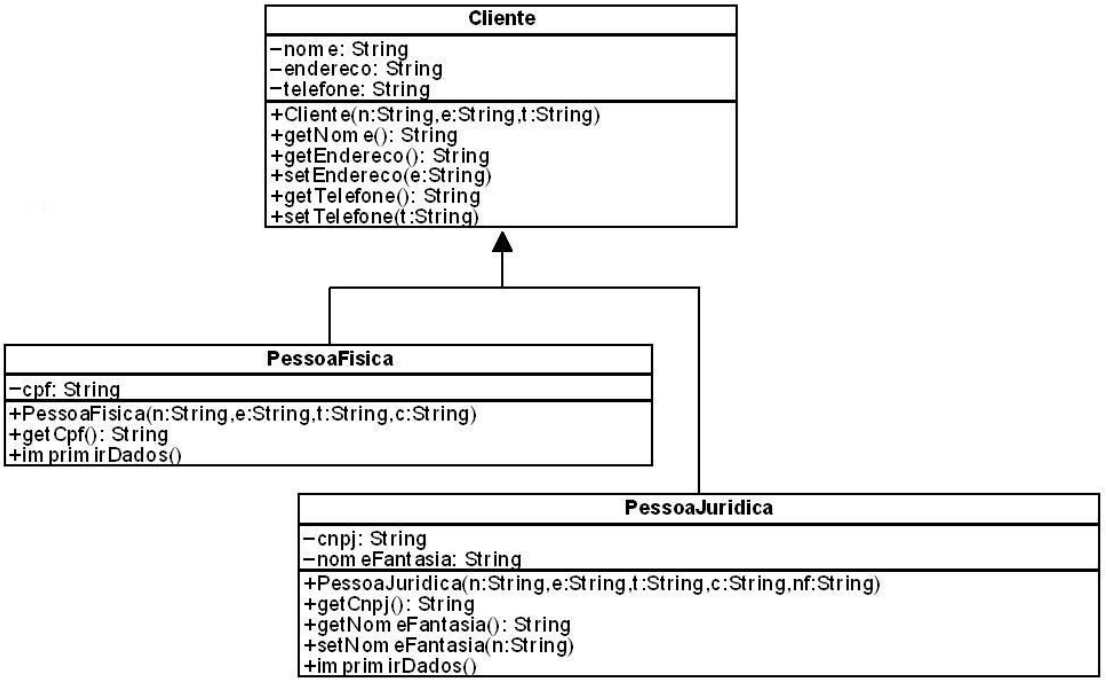
Questão Fácil (20%) - Clientes

Você é o responsável por desenvolver um sistema de cadastro de clientes para uma empresa. Essa empresa deseja guardar os nomes, endereços e telefones de cada cliente. Mas essa empresa possui dois tipos de clientes: pessoas físicas e pessoas jurídicas.

O cadastro dos clientes deve ser diferente para cada tipo de cliente. Se o cliente for uma pessoa física, deverá ser armazenado seu CPF juntamente com o restante dos dados. Já se o cliente for uma pessoa jurídica, deverá ser armazenado seu CNPJ e o nome fantasia da empresa do cliente.

A empresa deseja imprimir os dados dos clientes e saber a quantidade de clientes que ela possui cadastrado no sistema, tanto a quantidade geral como a quantidade de cada tipo de cliente.

Altere a modelagem a seguir e implemente as classes necessárias utilizando os conceitos de polimorfismo.



Questão Média (30%) - Conta Bancária

Vimos em aulas passadas que classes podem ser herdadas por outras. Vimos também todos os objetos em java herdam métodos de uma superclasse universal chamada Object. Dos métodos dessa classe, um dos mais interessantes é o toString. Procure este método na API do Java e responda em um .TXT qual seu tipo de retorno e se ele possui algum parâmetro.

A tarefa

Você foi contratado para implementar um sistema de gerenciamento em um Banco. Você foi pedido para elaborar uma classe ContaBancaria com os seguintes membros:

Atributos:

- String cliente;
- int num_conta;
- double saldo;

Métodos:

- sacar;
- depositar;
- toString;

Lembre-se de criar as limitações necessárias (sacar ou depositar um número negativo, deixar um saldo negativo, etc).

Ampliando a Classe

Agora acrescente ao projeto duas classes herdadas de ContaBancaria: ContaPoupanca e ContaEspecial, com as seguintes características a mais:

Classe ContaPoupança:

1. atributo int Dia_de_rendimento;
2. método calcularNovoSaldo, recebe a taxa de rendimento da poupança e atualiza o saldo.
3. toString;

Classe ContaEspecial

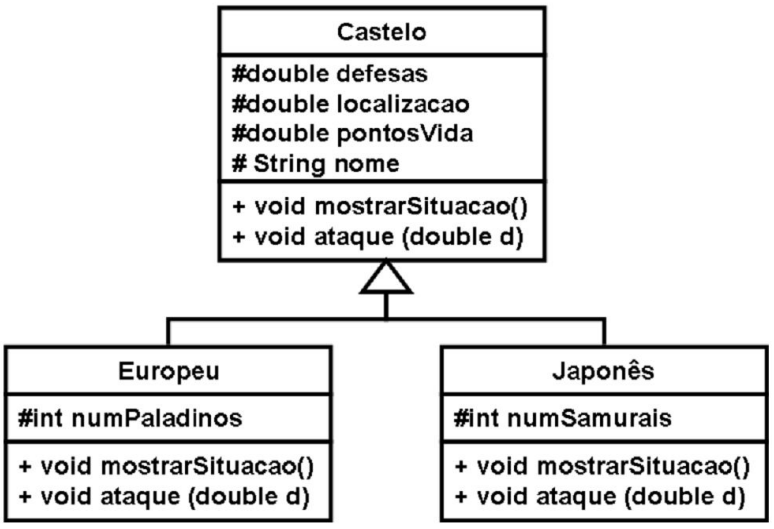
1. atributo double limite;
2. método sacar (permitindo saldo negativo até o valor do limite);
3. toString;

Escreva uma classe Banco que mantém uma coleção de contas bancárias. Sua classe deve possuir somente uma coleção. Lembre-se de adicionar métodos para manipular contas (Poupança e Especial) em sua classe Banco. Crie um método para imprimir os detalhes de todas as contas registradas no banco.

Questão Difícil (50%) - Jogo de estratégia

Criando Castelos

Veja o diagrama abaixo. Ele representa um sistema simples para representação de diversos castelos para um jogo de estratégia medieval. Nesse diagrama "**#**" representa visibilidade protegida (protected) e "**+**" representa público (public).



A tabela abaixo descreve como funcionam os métodos descritos nas classes. Escreva código de cada uma das classes desse diagrama usando a linguagem JAVA.

	void ataque(double d)	void mostrarSituacao()
Castelo	Se o campo defesas for > 0.0, então, reduzir o valor deste campo, senão reduzir o valor do campo pontosVida.	Mostrar os valores contidos nos campos da classe. Se pontosVida == 0.0, então, mostrar também a mensagem o castelo foi destruído.
Europeu	Se o número de Paladinos ou Samurais for >0, realizar um sorteio aleatório e eliminar entre 2 e 5 guerreiros. Senão, proceder como método ataque da classe Castelo.	
Japonês		

Crie classes de testes para cada uma das classes acima, incluindo testes positivos e negativos.

Criando personagens

De maneira similar aos castelos, crie uma hierarquia de classes para representar personagens do jogador no universo do Jogo. Nosso jogo permite ao jogador atuar como paladino ou samurai. Cada personagem deve conter pelo menos os seguintes atributos:

- **HP**: Hit Points determinam quanto dano o personagem pode receber antes de morrer. O HP pode ser recuperado através de magias, mas caso chegue a zero, o personagem será retirado do mapa e não poderá recuperá-los.
- **Força**: É a característica que determina quanto dano o personagem causa em um ataque físico. Cada personagem possui uma força baseada em suas características físicas.
- **Defesa**: Defesa é a característica responsável pela diminuição do dano recebido durante um ataque físico. Um ataque físico sempre causará, pelo menos, um ponto de dano.
- **Crítico**: Essa característica determina a chance (em porcentagem) do personagem causar dano extra ao realizar um ataque físico.

Personagens podem se movimentar pelo mapa do jogo, atacar uns aos outros, ou atacar castelos inimigos, desde que estejam em uma posição adjacente ao seu alvo.

Mapeando o jogo

O universo do jogo é representado através de uma matriz N x N, onde cada célula da matriz representa uma posição. Adicione uma classe para representar o mapa do jogo, incluindo métodos que permitam acessar/setar o valor de uma determinada posição.

Cada posição pode ser ocupada por um castelo ou por um personagem. Personagens são os únicos que podem se mexer. Você precisará adicionar uma classe pai á sua hierarquia de modo que todos os elementos do jogo possam estar representados no mapa. Explore o polimorfismo de métodos para implementar a funcionalidade de movimentação.

As mecânicas

A classe principal de qualquer game é a classe que contém o gameloop. A classe Game não será responsável por um loop e sim por gerir o nosso jogo com os métodos que você precisa criar. Essa classe será responsável, também, por armazenar as coleções e será ela quem irá criar um novo objeto quando necessário através de algum método gerador. Dicas: métodos como mostrar tabuleiro, mostrar recursos, numero de soldados, etc… Pense nisso como o **menu principal** do seu jogo.

Nosso jogo funcionará da seguinte maneira: Existirão dois castelos. Castelos serão representados na matriz tabuleiro como “E” para castelo Europeu e “J” para castelo japonês. Ambos devem ficar em posições distintas do mapa, só podendo ficar separadas através de, pelo menos, 3 casas em qualquer direção (diagonais não medíveis, ou seja, utilize linhas ou colunas). Os jogadores começarão com personagens randomicamente gerados pelo mapa. Cada nação começará com 3 “soldados” de cada lado. O controle será feito por linha de comando (procure na API por java scanner). Defina os botões de movimento como você preferir. Quando um soldado inimigo passar por cima do seu, o movimento não deve ser considerado, entretanto um ataque foi iniciado. Isso significa que, caso exista um personagem no endereço (3,3) dá matriz e um personagem em (4,3), caso o personagem de cima ande para baixo ou vice versa, um ataque deverá ser iniciado.

Resolução de combates

Para resolver um combate, precisamos nos ater aos atributos e aos estados dos personagens. Quando ocorrer, o jogador atacante irá lançar um dado d6 (devendo gerar um número aleatório entre 1 e 6). Caso o número seja menor ou igual à 2, o jogador vai ter errado o ataque e nada acontecerá. Caso o valor tenha sido 3, 4 ou 5, o jogador acertará um dano de acordo com a defesa do inimigo, isto é: (Valor do dado * valor de ataque - defesa do inimigo). Portanto, se temos 10 de ataque e o inimigo 20 de defesa, tirando 5 nos dados teríamos: (5 * 10 - 20 = 30) de dano. Reduzimos este valor dos pontos de vida do inimigo. Caso o inimigo fique sem pontos de vida, ele deverá ser removido da matriz tabuleiro. Caso o valor do dado seja 6, você causará *piercing* na armadura inimiga, portanto, multiplicará o dano por 3 e somará 15 ao valor do dano. Nesse caso, a armadura inimiga será descartada e o dano será todo projetado.

Exemplo de movimentação em um tabuleiro 4 por 4 (E = castelo europeu, J = castelo japonês, S = Samurai, P = Paladino).

```
E 0 0 P
0 0 0 0
```

```
0 0 S 0
0 J 0 0
```

É a vez do paladino, o jogador pressiona ‘s’ (indicando que está se movendo para baixo) e dá enter:

```
E 0 0 0
0 0 0 P
0 0 S 0
0 J 0 0
```

É a vez do samurai, o jogador pressiona ‘w’ (indicando que está se movendo para cima) e dá enter:

```
E 0 0 0
0 0 S P
0 0 0 0
0 J 0 0
```

Agora, caso o jogador que controla o paladino se mova para a esquerda, ele deverá entrar em combate.

A cada 5 movimentos, um jogador do seu time deve nascer randomicamente adjacente ao castelo de sua nação. Isso vai gerar continuidade ao jogo.

Vitória

Para vencer o jogo, o seu time deverá destruir o castelo inimigo. Os soldados devem sumir ao se mover em direção ao castelo inimigo, mas devem causar dano à estrutura de modo que 5 guerreiros sejam o suficiente para derrubá-lo.

Lembre-se

Estamos falando de um jogo: a liberdade criativa deve lhe guiar. Modifique as mecânicas e balanceie seu jogo como preferir. Veja que não falamos sobre seletor de jogador, como você vai saber qual jogar vai se movimentar nesta rodada? Fica à seu critério. Você pode fazer isso de maneira aleatória, de maneira sequencial ou até mesmo escolher qual de seus guerreiros deverá movimentar. Queremos ver sua capacidade de criar classes com baixo acoplamento (baixa dependência de outras classes), gerir coleções e utilizar herança e polimorfismo, portanto, as mecânicas são secundárias. Tente implementar o máximo que você viu em sala para fazer o jogo funcionar.

Obs: DOCUMENTE SEU CÓDIGO. Javadoc não é apenas opcional, ele é essêncial para manter a organização e a transparência do seu código. Código não documentado terá redução na pontuação.