UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
UNIDADE ACADÊMICA DE CIÊNCIA DA COMPUTAÇÃO

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Disciplina: Programação 2

Professores: Carlos Wilson/Francisco Neto

Período: **Turma:** **Data:** ___/___/___

Prova Final **NOTA FINALIZADA**

Assinatura do aluno:

ATENÇÃO: Instruções para realização da Avaliação da Aprendizagem

a) Verifique se a prova está completa. Caso contrário notifique aos professores em sala.

b) Você deverá assinar a folha com seu nome completo, legível no espaço destinado para a assinatura, utilizando caneta esferográfica azul ou preta.

c) Não serão permitidas rasuras nas questões de múltipla escolha.

d) Esta prova é individual. É vedada qualquer comunicação ou troca de material impresso entre os presentes, consulta a material bibliográfico, cadernos, anotações equipamentos eletrônicos ou outro de qualquer espécie.

e) Responda as questões de forma clara e objetiva.

f) Hora da prova NÃO é hora de tirar dúvidas do conteúdo com o professor.

g) A caligrafia deve ser nítida, clara e legível.

h) Antes de responder faça um rascunho ou um roteiro.

i) Leia a prova com calma e atenção. Todas as informações necessárias estão contidas nela e a interpretação faz parte da avaliação.

Boa prova!

1) Considere o código abaixo que implementa dois tipos de **carrinho de compras** que armazenam apenas os nomes dos produtos que serão comprados. Responda as seguintes questões: *(2,0)*

a) Nesse código existem dois erros de compilação. **Indique a linha** em que esses erros ocorrem e **explique porque** eles ocorrem **e qual a única modificação** que deve ser feita no código para corrigir esses erros? (1,0)

b) Quais os **valores impressos** nas linhas 38 e 39? (0,5)

c) Qual a diferença entre a impressão que será exibida ao executa a linha 41 para a que é exibida na linha 42. Justifique sua resposta de forma clara usando os elementos do código. (0,5)

```
01. public abstract class CarrinhoCompra {
02.
03.     private Collection<String> nomes;
04.     public CarrinhoCompra(){     }
05.
06.     public void imprimeCarrinho(){
07.             for(String nome : nomes){System.out.println(nome);}
08.     }
09.
10.     public void adiciona(String nome){this.nomes.add(nome);}
11.     public int qtdDeNomes(){return nomes.size();}
12. }

13. public class CarrinhoNormal extends CarrinhoCompra {
14.     public CarrinhoNormal(){
15.             super();
16.             super.nomes = new ArrayList<String>();
17.     }
18. }

19. public class CarrinhoDieta extends CarrinhoCompra{
20.     public CarrinhoDieta(){
21.             super();
22.             super.nomes = new HashSet<String>();
23.     }
24. }

25. public static void main(String[] args) {
26.     CarrinhoCompra carNormal = new CarrinhoNormal();
27.     carNormal.adiciona("presunto");
28.     carNormal.adiciona("banana");
29.     carNormal.adiciona("queijo");
30.     carNormal.adiciona("presunto");
31.
32.     CarrinhoCompra carDieta = new CarrinhoDieta();
33.     carDieta.adiciona("presunto");
34.     carDieta.adiciona("banana");
35.     carDieta.adiciona("queijo");
36.     carDieta.adiciona("presunto");
37.
38.     System.out.println(carNormal.qtdDeNomes());
39.     System.out.println(carDieta.qtdDeNomes());
40.
41.     carNormal.imprimeCarrinho();
42.     carDieta.imprimeCarrinho();
  }
```

2) Considere o código abaixo de uma classe de Exceção e a classe que a usa. Diante disso, responda: *(1,5)*

a) Esse código gera algum **erro de complicação**? Justifique sua resposta. *(0,5)*

b) Note que o método, no lugar de lançar uma *ArrayIndexOutOfBoundException*, captura-a e a lança novamente como uma *CompraInvalidaException*. Quais as **vantagens** e **desvantagens** (para o **código escrito** e o **design** do projeto) desse tipo de tratamento de Exception? *(1,0)*

```
public class CompraInvalidaException extends RuntimeException {

	public CompraInvalidaException(String mensagem){
		super(mensagem);
	}

	public CompraInvalidaException(){
		super("Erro ao recuperar produto do Carrinho.");
	}
}

public class Supermercado {
	private List<Produto> carrinho;
	....
	public Produto recuperaProduto(int indice) {
		try{
			return carrinho.get(indice);

		}catch(ArrayIndexOutOfBoundException exception){
			throw new CompraInvalidaException();
		}
	}
}
```

3) Você foi convidado(a) fazer uma implementação de um sistema para RPG de mesa (e.g. Dungeons & Dragons). A parte que você ficou responsável é a implementação das **habilidades** dos personagens. Para o seu projeto, existe a **habilidade de ataque** e a **habilidade de cura**. Todas as ações possuem nome. Além disso, em D&D existem vários tipos de dados com **diferentes faces**. São eles: d4, d6, d8, d10, d12, d20 que indicam até quanto o resultado do dado pode sair. No caso, de **1 a *x***, onde *x* pode ser, respectivamente, 4, 6, 8, 10, 12 e 20. Note que os dados vão determinar a **quantidade de dano/cura** da habilidade. Considerando que essas ações são *roláveis* (da expressão "rolar o dado", ou *roll the dice*), responda as seguintes perguntas: *(3,5)*

a) Como polimorfismo pode ser usado para implementar as **habilidades** e os **dados**? Responda com um diagrama de classes (apenas métodos, atributos e associações importantes) ou com uma **descrição textual clara**. (1,5)

b) Que **modificações** devem ser feitas no seu design para **incluir** uma **habilidade** de ação furtiva (*sneak action*)? E para **adicionar** a jogada de dois dados de 6 faces? Descreva textualmente e/ou apresente um novo diagrama de classes. (1,5)

c) Como podemos adicionar a classe **herói** que possui **um nome** e **começa com nenhuma habilidade**. O herói se comporta como uma **lista de habilidades**, de forma que ele deve aprender (adicionar), esquecer (remover) e imprimir as suas habilidades. Além disso, ele deve usar todas as habilidades que ele conhece. Qual o tipo de polimorfismo que deve ser utilizado para permitir que o Heroi tenha uma lista de habilidades? Justifique sua resposta de forma clara. (1,0)

4) Abaixo está a classe que representa um **nó de uma árvore binária**. Esse nó possui os seguintes atributos: uma **cor** que pode ser **preta** ou **branca**; um valor **inteiro**; e dois apontadores para seus filhos da **esquerda** e da **direita**. Um exemplo dessa árvore é apresentada na Figura 1. *(3,0)*

```
public class No {

	private int conteudo;
	private Cor cor;
	private No direita;
	private No esquerda;

	public int getConteudo() {return conteudo;}
	public Cor getCor() {return cor;}
	public No getDireita() {return direita;}
	public No getEsquerda() {return esquerda;}

	public No(int conteudo, Cor cor){
		this.conteudo = conteudo;
		this.cor = cor;
		this.direita = null;
		this.esquerda = null;
	}

	public boolean equals(Object obj) {
		if(obj instanceof No){
			No no = (No)obj;
			return no.getConteudo() == this.getConteudo();
		}
		return false;
	}
}

public enum Cor {PRETO,BRANCO;}
```

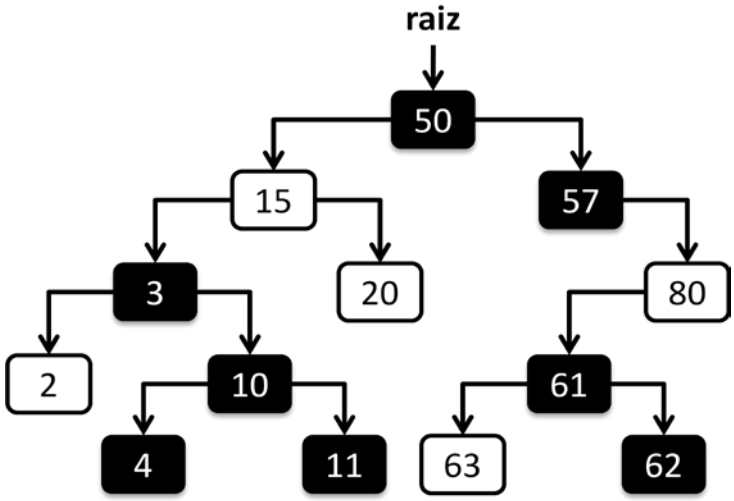

Figura 1 - Exemplo de árvore binária com Nós.

Baseado no código da classe **implemente os seguintes métodos recursivos**:

a) O método recursivo **imprimeCor(Cor c)** que realizar um caminhamento **pré-ordem** na árvore imprimindo o conteúdo (*int*) dos nós que são da cor especificada. Para o exemplo da Figura 1, a saída deve ser: (1,5)

```
raiz.imprimeCor(Cor.PRETO)
Preto 50
Preto 3
Preto 10
Preto 4
Preto 11
Preto 57
Preto 61
Preto 62
```

```
raiz.imprimeCor(Cor.BRANCO)
Branco 15
Branco 2
Branco 80
Branco 63
```

b) O método recursivo **qtdCor(Cor c)** que retorna um inteiro referente à quantidade de nós da cor especificada que pertencem à árvore. (1,5)

```
raiz.imprimeCor(Cor.PRETO): 8
raiz.imprimeCor(Cor.BRANCO): 4
```

Boa sorte e boa prova!