INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA PARAÍBA Campus Campina Grande	**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA – CAMPUS CAMPINA GRANDE**		
	CURSO:	**CURSO SUPERIOR ENGENHARIA DA COMPUTAÇÃO**	
	PERÍODO:		**TURMA:**
	DISCIPLINA:	**PROGRAMAÇÃO ORIENTADA A OBJETOS**	
	PROFESSOR:	**CÉSAR ROCHA VASCONCELOS**	SEMESTRE LETIVO

NOME:	NOTA:
	DATA: / /

PRÁTICA OBJETOS REMOTOS

1. Primeiramente, no *VirtualBox*, crie duas máquinas virtuais GNU/Linux idênticas (e.g., *Debian* ou *Xubuntu*, etc.) com conectividade de rede, para fins de simulação da comunicação entre um cliente e um servidor. Estas máquinas devem ter o *Java* devidamente instalado e configurado. Ao terminar, lembre-se de exportar estas duas máquinas, pois você precisará delas no dia da prova.
2. Crie um projeto no *Maven* chamado **proj-signos** e importe-o na sua IDE de preferência (*IntelliJ* ou *Eclipse*, por exemplo). Observe com atenção todos os requisitos funcionais abaixo e use fundamentos de Objetos Remotos e a API de RMI vistos em sala.
3. O objetivo desta prática é criar uma aplicação cliente-servidor. A partir de um signo (um texto simples) passado pelo cliente para o servidor, a aplicação deverá computar e retornar uma mensagem de signo associada como resultado. Parte da aplicação será executada no servidor, enquanto que a outra rodará na máquina-virtual-cliente. Lembre-se de que, em tempo de execução, a aplicação não deve ser executada "dentro" do *IntelliJ* ou *Eclipse*, mas, tão somente, em um terminal de linha de comando simples.
4. Primeiro, no UML, modele a interface **`ObjetoRemotoSignos_IF`**. Esta interface deve ter o seguinte método:
    a. **`String getMensagemSigno(String umSigno) throws RemoteException;`** Este método recebe como argumento um signo e deve retornar a mensagem de signo associada. Ainda no UML, indique qual o relacionamento é o mais adequado para relacionar **`ObjetoRemotoSignos_IF`** à interface **`java.rmi.Remote`** nativa da API do *Java;*
5. Ainda no UML, modele agora a classe chamada **`ServidorDeSignos`**. Esta classe deve ter os seguintes atributos e métodos:
    a. **`private Map<String,List<String>> mapaSignos;`** Este atributo deve armazenar uma coleção *mapa*, cuja chave é um signo e o valor da chave é uma lista com, pelo menos, duas respectivas mensagens de signo;
    b. **`public ServidorDeSignos();`** Este construtor deve popular todo o mapa, com os signos e suas respectivas mensagens.
    c. **`public String getMensagemSigno(String umSigno);`** **C**om base no signo que o usuário enviar pela rede para este servidor, este objeto remoto deve pesquisar no **`mapaSignos`** e retornar a respectiva mensagem para o usuário. Especificamente, este método deve sortear uma, dentre todas as mensagens na lista de mensagens associadas ao signo recebido, pela rede. Dica: use o método *random* da classe *Math* (ou até mesmo a classe *Random)* da API *Java* para gerar um número pseudo-aleatório e indexar, a esmo, uma mensagem existente na lista. Caso um signo inválido seja passado, o método deve retornar uma mensagem-padrão genérica, por exemplo, *"Signo não foi encontrado!";*
    d. Na modelagem, verifique qual relacionamento é o mais adequado no UML para relacionar a classe **`ServidorDeSignos`** à interface **`ObjetoRemotoSignos_IF`**.
    e. Conforme visto em sala, no *main*, deve-se: a) criar o "stub" e exportá-lo, corretamente; b) criar e cadastrar o "stub" no *java.rmi.registry.Registry*; e c) emitir uma mensagem de retorno no terminal, como, por exemplo, "Servidor de signos pronto..." ; Use a porta TCP 1099 (padrão).
6. No UML, modele agora a classe **`Cliente`** da aplicação. Esta classe deve ter os seguintes atributos e métodos:
    a. **`private static ObjetoRemotoSignos_IF stub;`** este atributo irá guardar o "stub" retornado pelo método lookup no *Registry:* **`registry.lookup("signos");`**
    b. **`private static Registry registry;`** este atributo irá guardar o objeto *Registry* retornado pelo método **`LocateRegistry.getRegistry(String hostServidor);`**
    c. **`public static executarCliente();`** este atributo irá inicializar as duas variáveis acima e ainda rodar um loop para permitir ao cliente pesquisar e enviar vários signos para o servidor remoto e obter as devidas mensagens de resposta; O *main* da classe **`Cliente`** deve chamar este método de classe para o cliente ser executado.
    d. Ainda no UML, indique qual o relacionamento é o mais adequado para relacionar **`Cliente`** aos tipos **`java.rmi.registry.LocateRegistry e java.rmi.registry.Registry`** nativos da API do *Java;*
7. Finalmente, após a modelagem, codifique os requisitos acima. Depois, copie apenas os arquivos necessários para a máquina cliente. Execute a aplicação servidora e o programa cliente em máquinas distintas! Todas as partes da aplicação cliente-servidor devem funcionar normalmente.