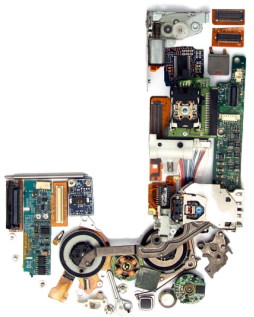
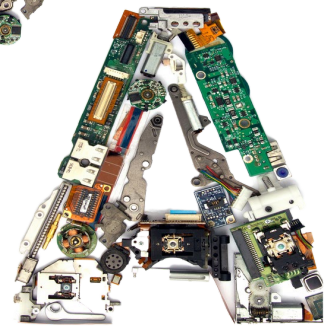
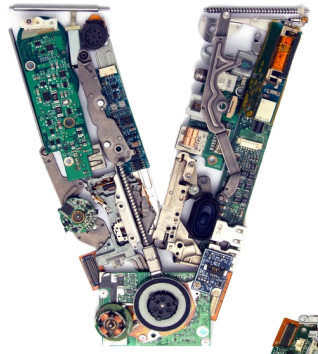
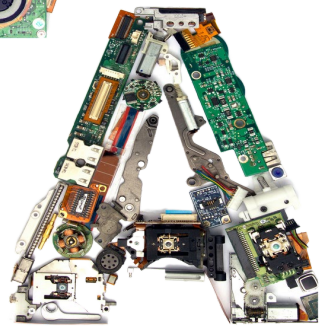
# Programação Orientada a Objetos com Java e WEB









## Exercício – Sistema de Controle de Estoque

# Sistema de Controle de Estoque

Desenvolver um sistema de **controle de estoque** para uma aplicação web utilizando **programação orientada a objetos em Java**, com **formularios HTML** para a interface de usuário e **banco de dados** para o armazenamento das informações. O sistema deve permitir a administração de usuários, produtos e vendas, além de realizar o controle de estoque de produtos.

**Requisitos do Sistema: o** sistema deve ser capaz de:

1. **Cadastrar Usuários:** com diferentes papéis: Administrador e Usuário Comum.
2. **Autenticar Usuários:** permitir o login com base no papel de cada um.
3. **Gerenciar Produtos:** cadastrar novos produtos, atualizar dados e controlar o estoque.
4. **Gerenciar Clientes:** cadastrar novos clientes e atualizar seus dados.
5. **Registrar Vendas:** um ou mais produtos podem ser vendidos em uma única venda.
6. **Listar Vendas:** permitindo ao administrador visualizar o histórico de vendas realizadas.

**Regras de Negócio:**

1. Apenas **administradores** podem acessar o **dashboard** para gerenciar usuários, clientes, produtos e realizar consultas de vendas.
2. O **estoque de produtos** deve ser atualizado automaticamente após o registro de uma venda.
3. **Vendas não podem ser realizadas** caso o estoque de um produto seja insuficiente.
4. **Senhas dos usuários** devem ser armazenadas de forma segura, utilizando **criptografia (bcrypt)**.

# Sistema de Controle de Estoque

Funcionalidades Específicas:

1. Cadastro de Usuários:
   - Apenas o **Administrador** pode cadastrar novos usuários.
   - O administrador deve escolher o **papel** do usuário (Administrador ou Usuário Comum).
   - As senhas dos usuários devem ser criptografadas e armazenadas no banco de dados.

2. Autenticação de Usuários:
   - O sistema deve permitir o login de usuários cadastrados.
   - Após o login, o sistema deve redirecionar o usuário com base no seu papel:
     - **Administrador**: Acesso ao **dashboard de administração**.
     - **Usuário Comum**: Acesso apenas à funcionalidade de **vendas**.

3. Gerenciamento de Produtos:
   - O administrador pode **cadastrar novos produtos**, atualizar informações de produtos existentes e controlar o estoque.
   - Cada produto deve conter os seguintes atributos: **id do produto**, **nome**, **quantidade em estoque**, e **preço**.

# Sistema de Controle de Estoque

4. Controle de Estoque:
   - Ao realizar uma venda, o estoque dos produtos vendidos deve ser automaticamente atualizado.
   - Não deve ser possível registrar uma venda caso a quantidade solicitada de um produto seja maior que a disponível em estoque.

5. Registro de Vendas:
   - O usuário comum pode registrar vendas, selecionando vários produtos.
   - Cada venda deve conter: cliente, produtos vendidos, quantidade de cada produto e data da venda.
   - O sistema deve verificar o estoque antes de registrar a venda.

6. Listagem de Vendas:
   - O administrador pode visualizar todas as vendas realizadas, com detalhes sobre os produtos vendidos, a data e o total da venda.

# Sistema de Controle de Estoque

Requisitos Técnicos:

1. O sistema deve ser desenvolvido em **Java**, utilizando **programação orientada a objetos**.
2. O sistema deve ter uma **interface web** baseada em **HTML** com **formulários** que permitam a interação com o usuário.
3. O **banco de dados** utilizado será o **Oracle** (ou outro banco relacional), e o sistema deve realizar operações CRUD (Create, Read, Update, Delete) nas tabelas do banco.
4. As seguintes tabelas serão utilizadas no banco de dados:
   a) **usuario**: armazena os dados dos usuários do sistema.
   b) **papel**: armazena os diferentes papéis dos usuários (administrador, usuário comum).
   c) **produto**: armazena os dados dos produtos.
   d) **cliente**: armazena os dados dos clientes.
   e) **fornecedor**: armazena os dados dos fornecedores.
   f) **venda**: armazena as informações de vendas realizadas, incluindo o cliente e a data da venda.
   g) **item_venda**: armazena os detalhes de cada item vendido em uma venda (produto, quantidade e subtotal).

# Sistema de Controle de Estoque

Regras de Implementação:

1. Programação Orientada a Objetos:
   - Utilize classes adequadas para representar entidades como Usuario, Produto, Cliente, Venda e Fornecedores.
   - As responsabilidades devem estar bem definidas entre as classes.

2. Formulários HTML:
   - Crie formulários HTML que permitam o cadastro de produtos, clientes e vendas.
   - Crie um formulário de login para autenticação dos usuários.

3. Banco de Dados:
   - Configure as tabelas no banco de dados Oracle.
   - Utilize JDBC para realizar as operações de inserção, consulta, atualização e exclusão.

4. Segurança:
   - Utilize bcrypt para criptografar as senhas dos usuários antes de armazená-las no banco de dados.
   - Garanta que apenas o administrador possa acessar a funcionalidade de gerenciamento de usuários e produtos, utilizando um filtro de servlet ou controle de sessão.

# Criação de Sequences (Oracle)

Uma **sequence** (sequência) é um objeto de banco de dados que gera números inteiros em uma ordem sequencial, o que é muito útil para criar valores exclusivos, como chaves primárias automáticas para tabelas. A sequência pode ser configurada para gerar números de acordo com várias regras, como incrementos, limites máximos e mínimos, entre outras.

A criação de uma **sequence** no Oracle é feita com o comando **CREATE SEQUENCE**. Você pode definir o valor inicial, o incremento, o valor máximo, etc. Exemplo de criação de uma **sequence**:

```
CREATE SEQUENCE seq_venda
    START WITH 1        -- O primeiro valor gerado será 1
    INCREMENT BY 1      -- Incrementa por 1 a cada novo valor
    NOMAXVALUE          -- Não há valor máximo
    NOCYCLE             -- Não reinicia a contagem quando alcançar o valor máximo
    CACHE 20;           -- Armazena 20 valores em cache para melhorar a performance
```

# Criação de Sequences (Oracle)

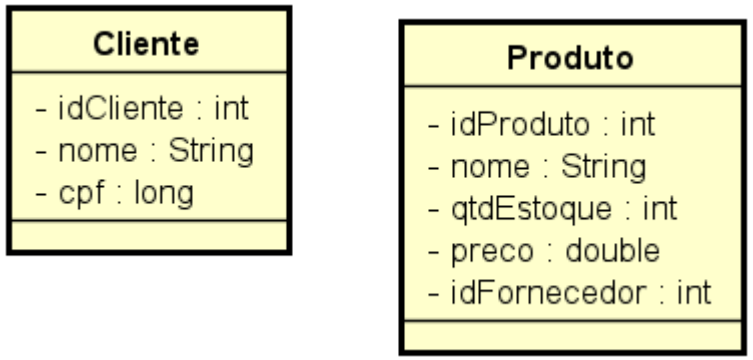
Para usar uma **sequence** e obter o próximo valor, você pode usar o comando **NEXTVAL**. Este comando retorna o próximo número da sequência. Exemplo de uso em um **INSERT**:

```
INSERT INTO venda (id_venda, id_cliente, id_vendedor, data_venda, total_venda)
    VALUES (seq_venda.NEXTVAL, 1, 2, SYSDATE, 1000);
```

Se você quiser consultar o último valor gerado durante a sessão, pode usar o **CURRVAL**:

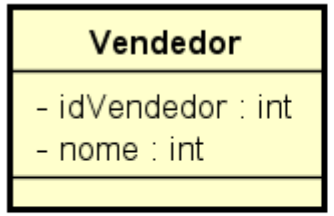
```
SELECT seq_venda.CURRVAL FROM dual;
```

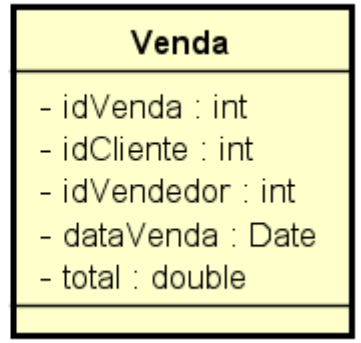
# Classes da Aplicação

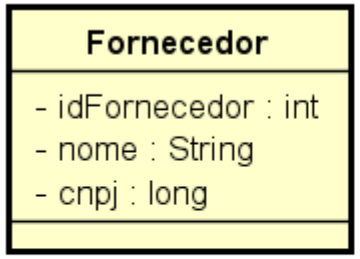


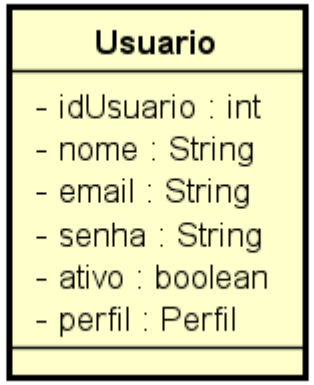


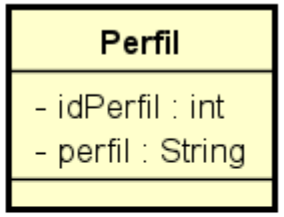




# Armazenamento de senhas em banco de dados

Armazenar senhas em texto puro em um banco de dados é uma prática extremamente perigosa e potencialmente devastadora para a segurança de qualquer sistema. Essa abordagem expõe informações críticas dos usuários a riscos desnecessários, pois, em caso de invasão, todas as senhas estarão facilmente acessíveis aos invasores.

Senhas armazenadas em texto puro permanece legível e compreensível, tanto para administradores do sistema quanto para qualquer pessoa que consiga acessar o banco de dados. Qualquer brecha de segurança, seja por meio de um ataque cibernético, um erro humano ou mesmo o uso indevido de privilégios, pode resultar na exposição de todas as senhas dos usuários. Com essas informações, um atacante pode não apenas comprometer contas individuais, mas também explorar a tendência dos usuários de reutilizar senhas, potencialmente ganhando acesso a outros sistemas e serviços.

Além disso, armazenar senhas em texto puro pode violar leis e regulamentações de proteção de dados, como a LGPD (Lei Geral de Proteção de Dados) no Brasil e o GDPR (Regulamento Geral sobre a Proteção de Dados) na União Europeia. Essas legislações exigem que informações pessoais e sensíveis sejam protegidas de forma adequada, e o armazenamento seguro de senhas é um requisito básico para a conformidade.

# Armazenamento de senhas em banco de dados

A melhor prática para proteger senhas é o uso de **funções de hash criptográficas**, que transformam as senhas em uma sequência de caracteres aparentemente aleatória e irreversível. Mesmo que um invasor consiga acessar o banco de dados, ele não terá acesso direto às senhas originais. Além disso, o uso de **salt**, um valor aleatório adicionado a cada senha antes de aplicar o hash, torna cada senha única, dificultando ainda mais a vida de quem tenta quebrar essa proteção.

## Aplicar funções de hash ou criptografia?

**Hashing**: é uma técnica usada para transformar dados (neste caso, uma senha) em um valor fixo (o **hash**) que é difícil de reverter. O **hash** é unidirecional, ou seja, ele é projetado para não ser convertido de volta para a senha original. Isso é ideal para verificar a autenticidade de uma senha, pois o sistema só precisa comparar o **hash** da senha fornecida com o **hash** armazenado. As funções de **hash** são irreversíveis por design. Ao aplicar um **salt** (um valor aleatório) a cada senha antes de hasheá-la, mesmo senhas idênticas resultam em hashes diferentes. Isso protege contra ataques de dicionário e tabelas rainbow.

**Criptografia**: é uma técnica usada para transformar dados em um formato ilegível que pode ser revertido (decriptado) para o formato original usando uma chave secreta. O objetivo principal da criptografia é proteger a confidencialidade dos dados em trânsito ou em repouso, mas de forma que eles possam ser recuperados.

# Tabelas Rainbow (texto gerado pelo ChatGPT)

**Tabela rainbow** é uma técnica utilizada em criptografia para acelerar o processo de quebra de hashes, facilitando a descoberta da senha original a partir de seu hash. Ela é uma forma otimizada de ataque de "pré-computação" que armazena um grande conjunto de hashes previamente calculados para combinações comuns de senhas e seus respectivos valores de hash.

## Funcionamento

**Pré-computação**: Antes do ataque em si, uma **tabela rainbow** é construída gerando hashes de muitas combinações possíveis de senhas (por exemplo, palavras comuns, senhas fracas, combinações de letras e números). Essas combinações são processadas por uma função de hash, e o resultado é armazenado na tabela junto com a senha correspondente.

**Encadeamento**: Para economizar espaço, em vez de armazenar todas as combinações de senha e hash, as tabelas rainbow utilizam um método chamado "encadeamento". Nesse método, são armazenados apenas o início e o final de uma sequência de transformação de hash. Com isso, é possível gerar um grande número de hashes a partir de um ponto inicial, reduzindo o tamanho da tabela.

**Busca**: Quando um invasor tem acesso a um hash que deseja quebrar, ele compara esse hash com os valores presentes na tabela rainbow. Se houver uma correspondência, o invasor segue o encadeamento da tabela até encontrar a senha original. Esse processo é muito mais rápido do que calcular hashes para cada possível combinação de senha em tempo real.

# Tabelas Rainbow (texto gerado pelo ChatGPT)

## Vantagens e Limitações

### Vantagens:

**Velocidade**: as tabelas rainbow aceleram significativamente o processo de quebra de hashes em comparação com um ataque de força bruta direto.

**Espaço Reduzido**: em comparação com as tabelas de hash tradicionais, as tabelas rainbow utilizam encadeamentos para reduzir o espaço necessário para armazenar combinações, tornando o ataque mais eficiente em termos de espaço.

### Limitações:

**Uso de Salt**: o uso de um salt (um valor aleatório adicionado a cada senha antes de aplicar o hash) para cada senha única torna as tabelas rainbow praticamente inúteis. Isso porque cada hash se torna único, mesmo que a senha seja a mesma.

**Memória e Tempo de Pré-computação**: criar uma tabela rainbow para um conjunto grande de combinações possíveis de senhas é um processo intensivo em tempo e memória.

# Tabelas Rainbow (texto gerado pelo ChatGPT)

## Prevenção Contra Ataques com Tabelas Rainbow

Para proteger sistemas contra ataques que utilizam tabelas rainbow, algumas boas práticas são recomendadas:
Uso de Salt: adicionar um salt único e aleatório a cada senha antes de aplicar a função de hash torna impossível o uso de tabelas rainbow pré-calculadas.

Funções de Hash Lentamente Computáveis: usar funções de hash que são computacionalmente mais custosas, como bcrypt, scrypt ou argon2, reduz a viabilidade de ataques baseados em pré-computação.

Senhas Fortes: Incentivar os usuários a usarem senhas fortes e complexas aumenta o tamanho do espaço de busca necessário para construir uma tabela rainbow, tornando o ataque inviável.

# Algoritmo de hash PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) é um algoritmo utilizado para derivar uma chave criptográfica a partir de uma senha. Ele é amplamente utilizado em sistemas de segurança para proteger senhas armazenadas, oferecendo uma solução resistente a ataques de força bruta e ataques de dicionário.

O algoritmo PBKDF2 foi originalmente especificado como parte do padrão PKCS #5 (Public-Key Cryptography Standards), que é um conjunto de normas desenvolvidas pela RSA Laboratories. O objetivo dos padrões PKCS é fornecer diretrizes para implementar diversos aspectos da criptografia de chave pública.

A primeira versão do PKCS #5 foi publicada em 1993, e a versão que introduziu o PBKDF2 foi a PKCS #5 v2.0, lançada em setembro de 2000. Esse padrão foi desenvolvido para fornecer uma maneira segura de derivar chaves criptográficas a partir de senhas, especialmente considerando que senhas humanas geralmente são fracas e mais fáceis de adivinhar ou atacar do que chaves criptográficas geradas aleatoriamente.

# Algoritmo de hash PBKDF2 – características

**Derivação de chave segura**: O algoritmo usa uma senha, um **salt** (um valor aleatório exclusivo para cada senha) e repetições de uma **função hash criptográfica** para gerar uma chave. Isso impede que **ataques pré-calculados**, como **ataques de tabelas de hash** (**rainbow tables**), sejam eficazes.

**Repetições controláveis** (**iterações**): O algoritmo realiza repetidamente uma função hash (como SHA-256, SHA-512, etc.) por um número especificado de iterações, aumentando a dificuldade de um atacante calcular a chave derivada a partir de uma senha. Um número alto de iterações torna o processo mais lento para o atacante, embora a diferença no tempo de processamento seja minimamente perceptível para o usuário legítimo.

**Salt**: O uso do **salt** (um valor aleatório) adicionado à senha antes de gerar a chave é fundamental para garantir que senhas iguais não resultem na mesma chave derivada, dificultando ataques de força bruta que tentam reutilizar chaves pré-computadas.

**Chave de saída variável**: O algoritmo permite especificar o tamanho da chave derivada. Isso o torna útil para aplicações como criptografia de discos, onde é necessário gerar chaves de tamanhos variados.

# Algoritmo de hash PBKDF2 – etapas

```java
// Número de iterações (pode ser ajustado)
private static final int ITERATIONS = 10000;

// Tamanho da chave gerada
private static final int KEY_LENGTH = 256;

// Algoritmo utilizado
private static final String ALGORITHM = "PBKDF2WithHmacSHA256";
```

PBKDF2WithHmacSHA1: Utiliza o algoritmo SHA-1 como a função de hash.

PBKDF2WithHmacSHA512: Utiliza o algoritmo SHA-512 como a função de hash, que é mais seguro do que SHA-1.

PBKDF2WithHmacSHA384: Utiliza o algoritmo SHA-384 como a função de hash, uma variação de SHA-512.

Essas substituições controlam a função de hash utilizada pelo PBKDF2. Se você estiver buscando mais segurança, SHA-256 ou SHA-512 são geralmente recomendados sobre SHA-1, que é mais antigo e menos seguro.

# Algoritmo de hash PBKDF2 – etapas

```java
// Gerar um salt aleatório
public static byte[] getSalt() {
    SecureRandom sr = new SecureRandom();
    byte[] salt = new byte[16];
    sr.nextBytes(salt);
    return salt;
}
```

O **salt** é um valor aleatório que é utilizado junto com a senha antes de aplicar um algoritmo de **hash**. Sua principal função é garantir que senhas idênticas gerem hashes diferentes, aumentando a segurança contra ataques como tabelas de pré-computação (tabelas arco-íris) e ataques de força bruta.

O método **getSalt()** gera um salt de 16 bytes (128 bits) usando o gerador de números aleatórios criptograficamente seguro **SecureRandom** (**pacote java.security.SecureRandom**). Ele retorna esse salt como um array de bytes, que pode ser usado em operações de derivação de chaves ou hashing, como no PBKDF2. O uso de **SecureRandom** garante que o salt seja imprevisível e resistente a ataques de força bruta.

Esse salt gerado pode ser armazenado junto com o hash da senha e é essencial para garantir que cada senha tenha um hash único, mesmo que duas pessoas utilizem a mesma senha.

# Algoritmo de hash PBKDF2 – etapas

A classe **PBEKeySpec** especifica os detalhes para a geração da chave derivada da senha. A combinação dos parâmetros cria uma especificação para o algoritmo PBKDF2 processar a senha.

As senhas em Java são geralmente convertidas para um array de caracteres (char[]) porque isso oferece uma forma mais segura de lidar com senhas em memória. Arrays de caracteres podem ser imediatamente apagados da memória após o uso, enquanto strings são imutáveis e permanecem na memória até serem coletadas pelo **garbage collector**.

```
public static String codificar(String senha, byte[] salt) {
    char[] senhaEmChar = senha.toCharArray();
    PBEKeySpec spec = new PBEKeySpec(senhaEmChar, salt, ITERATIONS, KEY_LENGTH);
    try {
        SecretKeyFactory skf = SecretKeyFactory.getInstance(ALGORITHM);
        byte[] hash = skf.generateSecret(spec).getEncoded();

        // Codifica o hash em Base64
        return Base64.getEncoder().encodeToString(hash);
    } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
        throw new RuntimeException("Erro ao gerar o hash da senha", e);
    }
}
```

O método **generateSecret**(**spec**) utiliza a especificação definida no **PBEKeySpec** para derivar a chave secreta (hash) da senha. O método **getEncoded()** converte essa chave em um array de bytes, que é o hash gerado pela função PBKDF2.

A classe **SecretKeyFactory** é utilizada para criar chaves secretas baseadas em algoritmos criptográficos. O método **getInstance**(**ALGORITHM**) obtém uma instância de **SecretKeyFactory** configurada para o algoritmo especificado, que neste caso é definido pela constante **ALGORITHM**, provavelmente "PBKDF2WithHmacSHA256".

O hash, que está em formato de array de bytes, é convertido para uma string em formato **Base64**. O **Base64** é uma maneira comum de representar dados binários em formato de texto legível, o que facilita o armazenamento do hash em bancos de dados ou arquivos de texto. A string resultante é retornada como o valor final do método.