SLIDE 1 – Revisão Orientação a Objetos

Orientação a Objetos

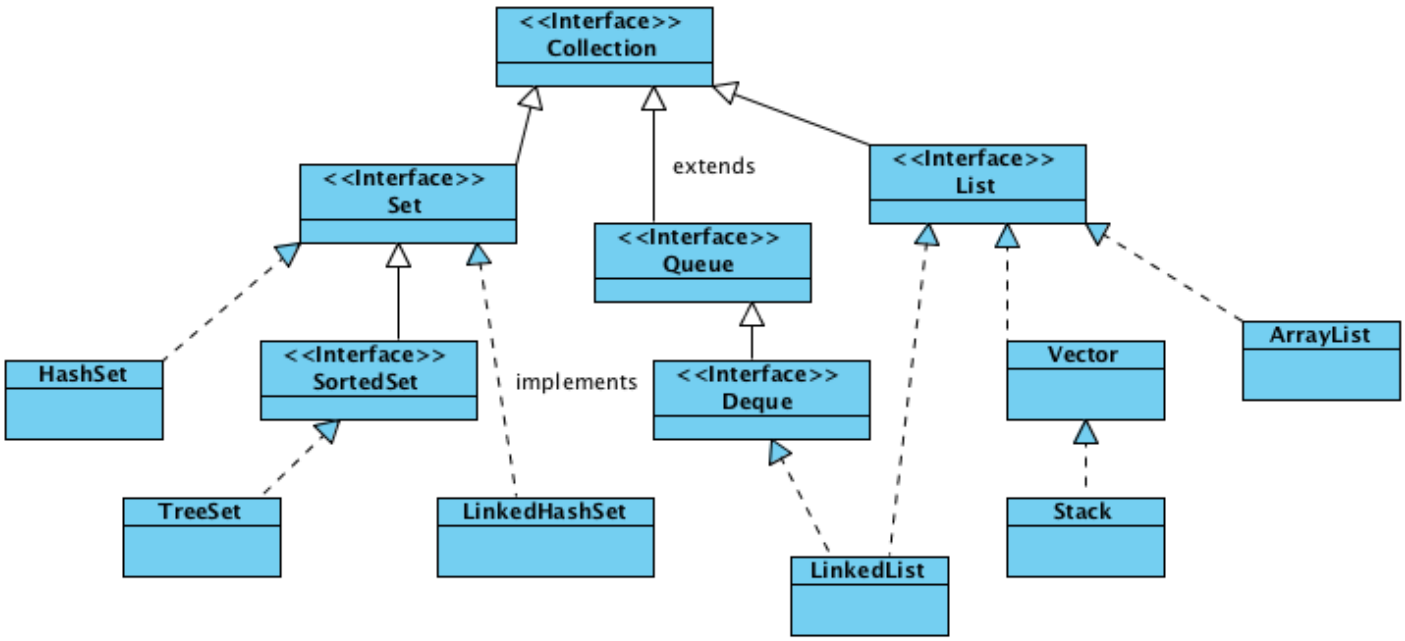
- Objeto
- Domínio da Aplicação
- Abstração
- Classe
    - Instância
    - Atributos
    - Operações
- Visibilidade
    - Público (+)
    - Protegido (#)
    - Privado (–)
    - Pacote (~) – Somente classificadores declaramos no mesmo pacote podem usar a característica
- Encapsulamento
    - setter
    - getter
- Método Construtor
    - Com argumento
    - Sem argumento
- Relacionamento
    - Associação
    - Agregação
    - Composição
    - Dependência
        - *Quando uma classe recebe um objeto de outra classe como parâmetro, uma classe acessa o objeto global da outra. Nesse caso existe uma dependência entre estas duas classes, apesar de não ser explícita.*
        - *Linha tracejada com seta na ponta*
    - Generalização – Herança

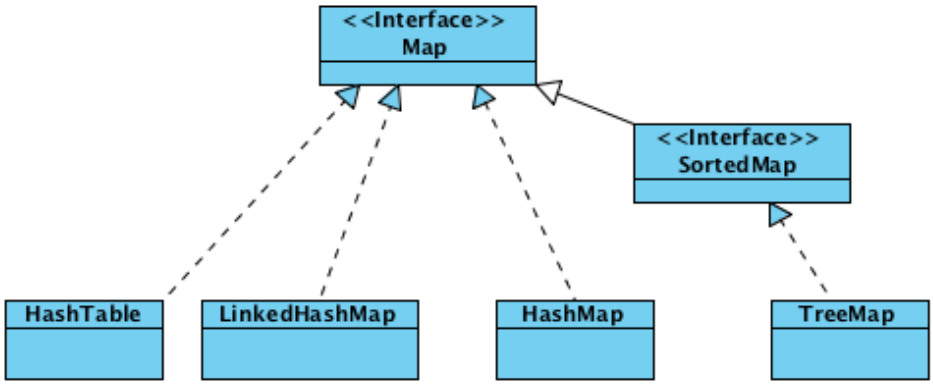
- Classes Abstratas
    - Os nomes das classes abstratas são escritas em itálico
- Interfaces
- Polimorfismo

SLIDE 3 – Collections

Collections

- As classes que implementam as estruturas de dados foram adicionadas ao Java a partir da versão 1.2 (Vector e HashTable)
- O Framework Java Collections foi inserido ao Javana versão Java 5





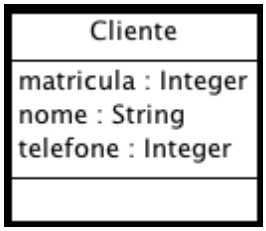
- **Listas** (List)
  - São estruturas lineares de armazenamento
  - Características:
    - Possuem um início e um fim
    - Permitem operações em qualquer posição
    - Não são permitidas lacunas entre os elementos
    - Podem possuir disciplinas de acesso
  - Em estrutura de dados
    - Listas sequenciais (arrays);
      - **ArrayList**
    - Listas encadeadas;
      - LinkedList
  - **ArrayList**
    - Possui um Array encapsulado como estrutura de dados.
    - Permite acesso indexado
    - Possui deslocamento em operações
      - Início; e
      - Posições específicas

        - Possui tamanho inicial definido (“fixo”);
            - Inicia com 10 posições e à medida que a quantidade se aproxima do valor total, um novo array é criado com tamanho maior
            - Todos os elementos do array original são copiados para o novo array.
    - **LinkedList**
        - Encapsula uma lista duplamente encadeada com descritor;
        - Não possui acesso indexado;
            - Para acessar uma posição específica é necessário varrer a lista a partir da referência início.
        - Para operações na lista são feitas apenas trocas de apontamento das referências;
        - Não possui tamanho inicial (“fixo’).
        - Todos os métodos que o ArrayList implementa, também são implementados pelo LikedList, ou seja, podem ser utilizados na mesma forma;
- **Set** (Conjuntos)
    - Os conjuntos são estruturas que não permitem elementos repetidos.
    - Eles não possuem características lineares.
    - Podem ser implementos a partir de:
        - Tabelas Hash; ou
        - Árvores
            - Somente a implementação na forma de árvore pode ser ordenada
    - **HashSet**
        - Conjunto implementado como Hash
            - Utiliza uma lista sequencial como tabela de endereços
            - Utiliza uma função para fazer o “espelhamento” dos valores;
            - Utiliza uma lista encadeada para tratar colisões
            - Não mantem a ordem de inserção dos valores;
        - Possui um excelente tempo de inserção dos valores;
        - Indicado para acesso a dados pelo seu valor;
        - Para operações de “localizar”;
        - Estrutura muito utilizada pelos Sistemas de Gerenciadores de Banco de Dados (SGBD);
    - **TreeSet**
        - O uso dos métodos apresentados no HashSet são ou mesmos para o TreeSet
        - O uso da interface SortedSet, já garante a ordenação na forma crescente.
- **Map** (Mapas)
    - Estruturas de dados baseado na combinação:
        - Chave -> valor
    - Onde
        - Para cada valor deve-se ter uma chave diferente
        - Elementos com a mesma chave são sobrescritos
    - Podem ser implementadas utilizando:
        - Hash; e
        - Árvores
    - Pode-se utilizar como chave e valor, qualquer tipo de Objeto.
    - Para tipos abstratos de dados é necessário:
        - Implementar o método enquals e hashcode;
    - Indicados para:
        - Recuperação de elementos por uma chave
        - Acesso muito rápido à elementos
    - É possível:
        - Pode-se recuperar as chaves (na forma de um conjunto)
        - Pode-se recuperar apenas os vaores (na forma de uma lista)
    - **HashMap**
        - É um mapa utilizando a estrutura de dados Hashing

        - Recebe todas as características de uma implementação de Hash com as ações esperadas para um mapa.
    - **TreeMap**
        - Recebe todas as características de uma árvore
        - Permite todas as operações de um mapa;
        - Sua implementação e utilização para ordenação é muito similar a do TreHash
        - Indicado:
            - Para mapas em que precisa-se de ordenação das chaves

SLIDE 4 – Collections + Tratamento de exceções

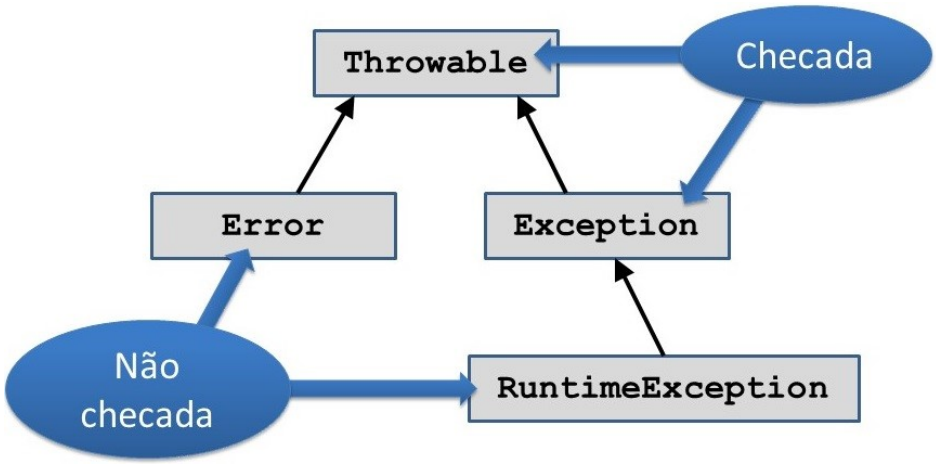
Igualdade em Hash

- Para entendermos como é possível identificar se dois elementos (objetos complexos) são iguais e devem ser ou não adicionados em um Hash, formulou-se a seguinte classe:
  - 

- Para adicionar um cliente (ou qualquer outro tipo de objeto complexo) em um hash, é necessário implementar dois métodos:
  - hashCode; e
  - equals
- Esses métodos devem ser implementados na classe do objeto complexo, no nosso caso na classe Cliente.
- O mesmo comportamento deve ser implementado, quando se deseja qualquer um objeto complexo, seja chave de um Mapa.
- Da mesma forma que um Mapa não pode possui duas chaves iguais, também é necessário adicionar os métodos ***equals*** e ***hashCode*** ao objeto chave.

Tratamento de Exceções

- O que é uma exceção?
  - **Exceção** é um evento que ocorre fora do padrão de execução esperado;
  - São muito utilizadas pelos Sistemas Operacionais e demais sistemas;
  - Usar tratamento de exceção permite detectar erros e manipular esses erros, ou seja, trata-los;
- Blocos para tratamento de exceções
  - **try**:
    - É usado para indicar um bloco de código que possa ocorrer uma exceção.
  - **catch**:
    - Serve para manipular as exceções, ou seja, tratar o erro.
  - **finally**:
    - Sempre será executado depois do bloco try/catch;
    - É importante saber que esse bloco sempre será executado;
    - Sempre que se trabalhar com a abertura de um recurso (arquivo, conexão,...) deve-se fechar a mesma aqui.
- **Visão hierárquica**



  - **Exception**
    - As classes que deveriam aqui lançar exceções e não erros de programação.

        - Exemplo: tentar abrir um arquivo que não existe. Então, é lançada uma exceção verificada, porque a classe de leitura de arquivos deriva de ***Exception***.
    - **Runtime Exception**
        - São exceções que indicam erros de programas (não de lógica, pois senão não passaria pelo compilador).
        - Esse tipo de exceção é conhecida como não checada. Sendo assim, não é requisito declarar uma cláusula try{} e catch{}. Ex.: tentar converter "dois" em "2".
    - **Error**
        - Causadas por condições externas à aplicação, e normalmente a aplicação não tem como antecipar-se e tratar.
            - ***Exemplo***: a aplicação abre um arquivo para leitura, mas não consegue gravar por uma falha de hardware ou mal funcionamento do sistema. Isso causará o lançamento de java.io.IOError.
        - Não são exceções, e sim erros que jamais poderiam ter acontecido.
            - ***Exemplo***: estouro da memória.
    - **Exceções checadas**
        - Há situações excepcionais que uma aplicação bem escrita deve antecipar e tratar.
        - Exceções checadas estão sujeitas ao *Catch* or *Specify Requirement*.
        - Todas exceções são checadas, exceto aquelas indicadas por ***Error*** e ***RuntimeException*** e suas especializações.
    - **Exceções não checadas**
        - Causada por condições internas à aplicação e normalmente a aplicação não tem como recuperar-se ou tratar.
            - Geralmente indica problema no desenvolvimento: se um erro de lógica permite, por exemplo, a chegada de uma referência nula ao construtor de ***FileReader***, ***NullPointerException*** será lançada.
        - O desenvolvedor pode capturar a exceção (catch), mas faz mais sentido corrigir o erro.
- **Exceções checada X não checada (Quando utilizar)**
    - Exceções não checadas representam o resultado de um problema de programação do qual não há como lidar ou recuperar de maneira alguma.
        - ***Exemplos***: exceções aritméticas, referência nula e erro de indexação (tentativa de acessar um índice que não existe).
    - Não lance ***RuntimeException*** ou uma especialização dela só porque não quer se incomodar com a especificação de exceções que podem ser lançadas.
    - Se é razoável que um cliente receba a exceção para tratá-la, use exceção checada.
        - Se, no entanto, não tiver o que fazer a respeito do erro, faça-a não checada.
- **Erros comuns no uso da Exceptions**
    - catch (Exception e) {}
        - Nunca deixe o bloco catch vazio!
    - Continuação do fluxo de execução após erro
        - Analise sempre até que ponto o fluxo deve ser interrompido quando ocorrer uma exceção.
    - Falha relatada incorretamente
        - Verifique com cautela a exceção capturada, tenha certeza do que será feito log.
- **Boas práticas (onjava.com)**
    - Sempre feche recursos após usá-los
    - Não use exceções para controlar fluxo
    - Não ignore ou “mate” a exceção
    - Não capture exceções de primeiro nível:
    - Fazendo isso, capturam-se todas as exceções, inclusive RuntimeExceptions.

```
try{
..
}catch(Exception ex){
}
```

- Fazendo isso, capturam-e todas as exceções, inclusive ***RumtimeExceptions***.

SLIDE 5 – Manipulação de Arquivos

Manipulação de arquivos

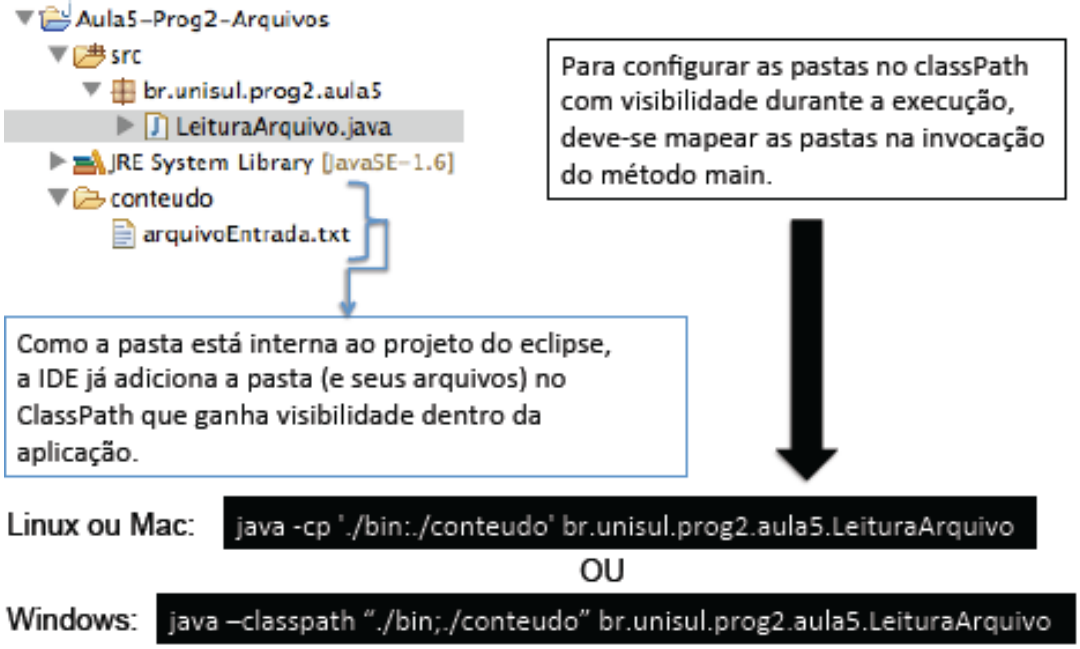
- Diariamente utilizamos arquivos em nossos computadores, para armazenar e consultadas dados.
- Este é um recurso bastante importante em programação, pois permite que dados sejam persistidos ao final da execução de um programa.
- Até então mantínhamos os dados de nossas aplicações em memória, de modo que quando a aplicação fosse encerrada os dados eram perdidos.
- A persistência de dados em arquivo, pode ser um alternativa para o problema apresentado.

Pacote ***java.io***

- O Java trata a entrada e saída como fluxos de dados (os tão chamados ***Streams***);
- As classes ligadas a **io** estão nos pacotes ***java.io*** e ***java.nio***;
- Instâncias da classe ***java.io.File*** representam caminhos (paths) para possíveis locais no sistema operacional.

```
//Declarando e instanciando um arquivo
File arquivo = new File("conteudo/arquivoEntrada.txt");
```

- **Classe File**
  - Essa classe está diretamente ligada à um caminho;
  - Esse caminho pode existir ou não;
  - Através deste objeto é possível verificar propriedades de um arquivo ou pasta.
    - Mas ele não interage com o seu conteúdo.
  - É possível também criar arquivos e pastas, mas não escrever dentro do mesmo.
- **ClassPath**

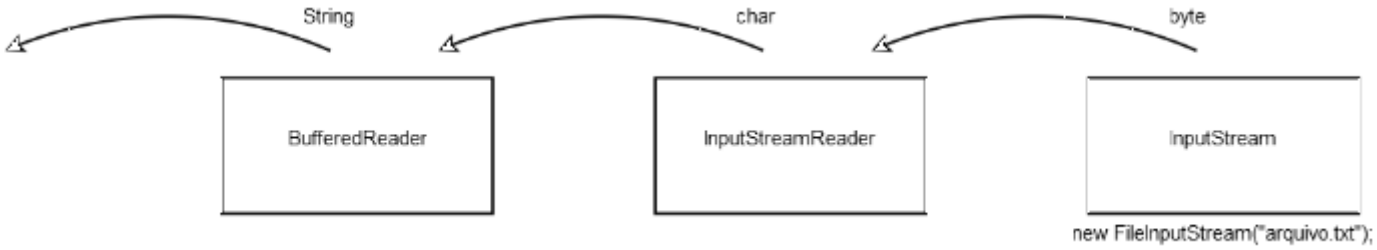


- Criando subpasta
- Criando arquivo

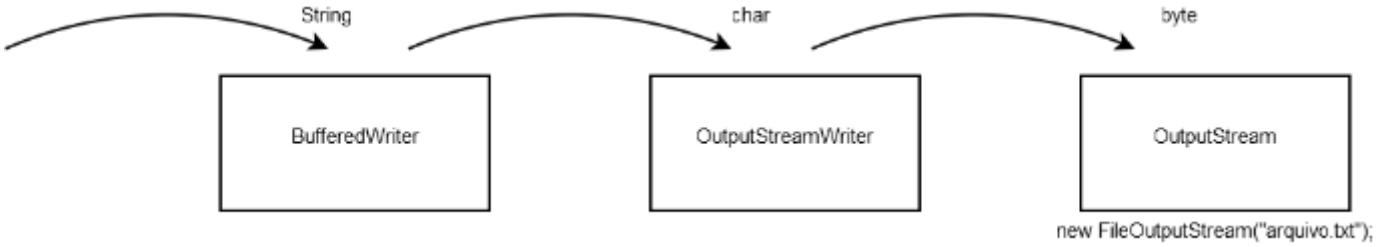
Lendo conteúdo de arquivos

- **Lendo arquivo**
  - A classe **`InputStream`**, é uma classe abstrata (não pode ser instanciada);
  - Para lermos de um arquivo físico, deve-se utilizar a classe concreta **`FileInputStream`**;

- Quando trabalhamos com *java.io*, diversos métodos lançam **`IOException`**, que é uma exception do tipo checada - o que nos obriga a tratá-la ou declará-la.
  - `InputStream` tem diversas outras filhas, como `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, entre outras.
- Para recuperar um caractere, precisamos traduzir os bytes com o encoding dado para o respectivo código unicode, isso pode usar um ou mais bytes.
- Escrever esse decodificador é muito complicado, quem faz isso por você é a classe **`InputStreamReader`**.
  - **`InputStreamReader`** é filha da classe abstrata ***`Reader`***, que possui diversas outras filhas - são classes que manipulam chars.
- Apesar da classe abstrata **Reader** já ajudar no trabalho de manipulação de caracteres, ainda seria difícil pegar uma String.
- A classe **`BufferedReader`** é um ***`Reader`*** que recebe outro ***`Reader`*** pelo construtor e concatena os diversos chars para formar uma String através do método ***readLine***:
  - Como o próprio nome diz, essa classe lê do **`Reader`** por pedaços (usando o buffer) para evitar realizar muitas chamadas ao sistema operacional. Você pode até configurar o tamanho do buffer pelo construtor.
- Na prática:
  - **`InputStream`** lê o arquivo na forma de byte,
  - o **`InputStreamReader`** converte de byte para char
  - e por fim **`BufferReader`** transforma o conteúdo para String:



- **Escrevendo em Arquivo**
  - Da mesma forma que para leitura temos 3 etapas, para a escrita não é diferente.
  - Escrita dos bytes:
    - **`OutputStream`**
  - Conversão para caracteres:
    - **`OutputStreamWriter`**
  - Trata valor literal:
    - **`BufferedWriter`**



  - É possível escrever apenas no final do arquivo, sem sobrescrevê-lo?
    - Sim. Basta adicionar o parâmetro true, na construção do objeto da classe FileOutputStream.

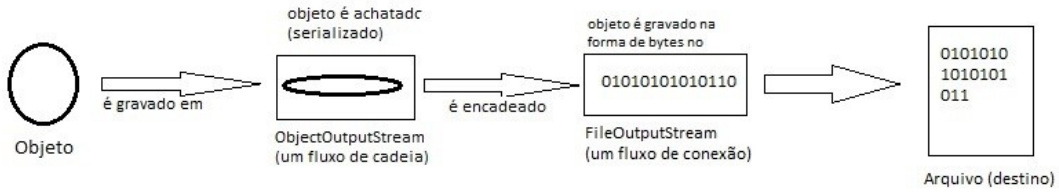
SLIDE 5 – Serialização e a manipulação de arquivos

Serialização

- **O que é serialização?**

- A serialização resume-se em um único objetivo: salvar, gravar e capturar o estado de um objeto.
- Ou seja, tenho um objeto de uma classe e quero salvar seu estado, então uso serialização, pois quando quiser usar esse objeto com seu último estado gravado é somente deserializá-lo.

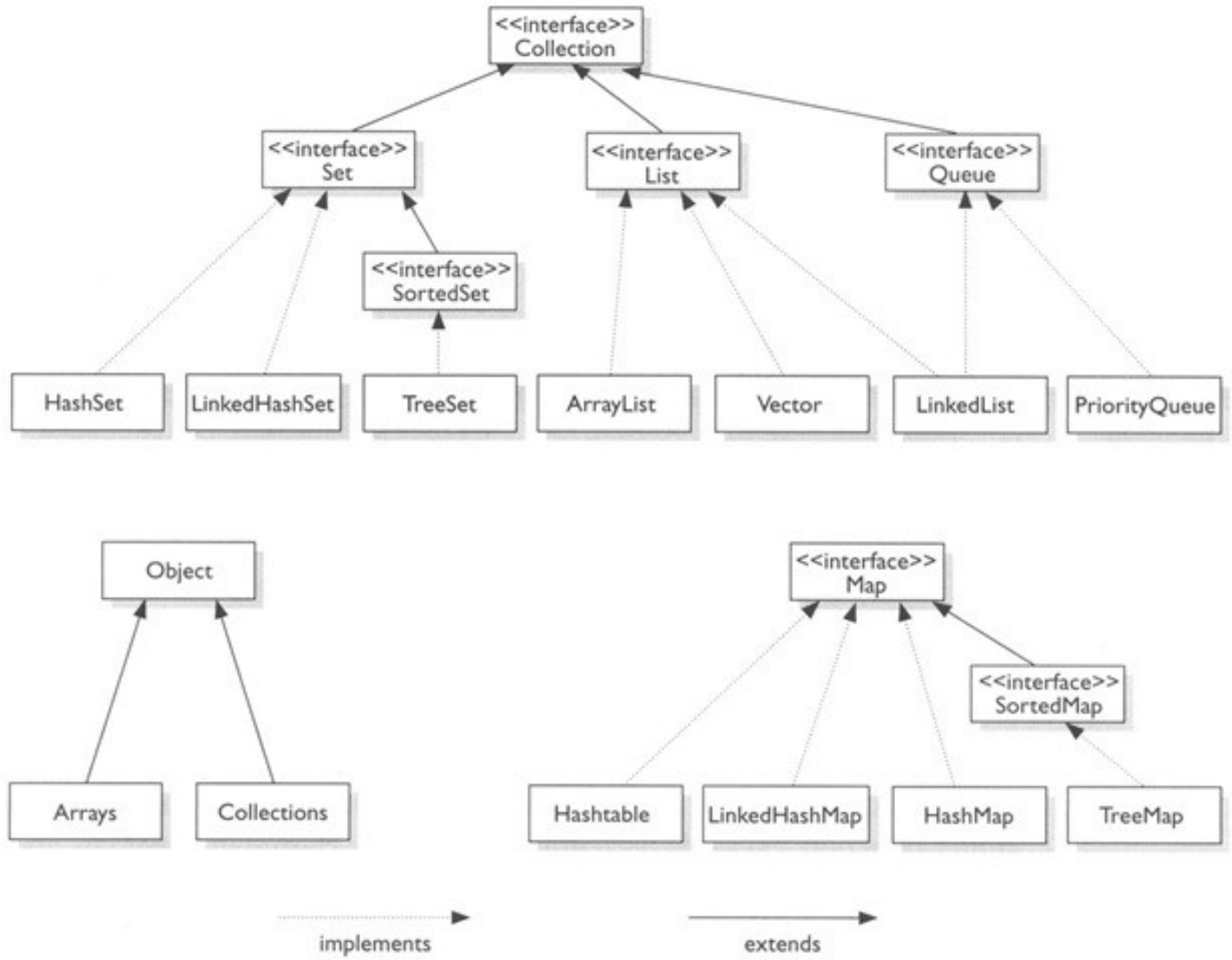
- **Mas por que usar serialização?**
  - A partir do momento que se serializa um objeto ou uma variável de instância, seu valor é gravado (em um arquivo), armazenado e futuramente posso obter esse valor através da deserialização



- **Serialização em Java:**
  - Para que um objeto seja serializado em Java é necessário que o mesmo implemente a interface **`Serializable`**.

ESTRUTURA DE DADOS - SLIDE 17 – Collections Java

Collections Java



- **List** *(interface)*
  - Interface que estende Collection
  - Define Coleções Ordenadas (sequencias), onde se tem o controle total sobre a posição de cada elemento, identificado por um índice numérico.
  - **ArrayList** *(implements List)*
    - Implementação de List que utiliza internamente um *array de objetos*
    - Em uma inserção onde o Tamanho do array interno não é suficiente, um novo array é alocado (de tamanho igual a 1.5 vezes o array original), e todo o conteúdo é copiado para o novo array.
    - Esta implementação é a recomendada quando o tamanho da lista é previsível (evitando realocações) e as operações de inserção e remoção são feitas, em sua maioria, no fim da lista (evitando deslocamentos), ou quando a lista é mais lida do que modificada (otimizado para leitura aleatória).
  - **LinkdList** *(implements List)*
    - Implementação de List que utiliza internamente uma *lista encadeada*.
    - A localização de um elemento na n-ésima posição é feita percorrendo-se a lista da ponta mais próxima até o índice desejado.
    - A inserção é feita pela adição de novos nós entre os nós adjacentes, sendo que antes é necessária a localização desta posição
    - Esta implementação é recomendada quando as modificações são feitas em sua maioria tanto no início quanto no final da lista, e o percorrimento é feito de forma sequencial (via Iterator) ou nas extremidades, e não aleatória (por índices)

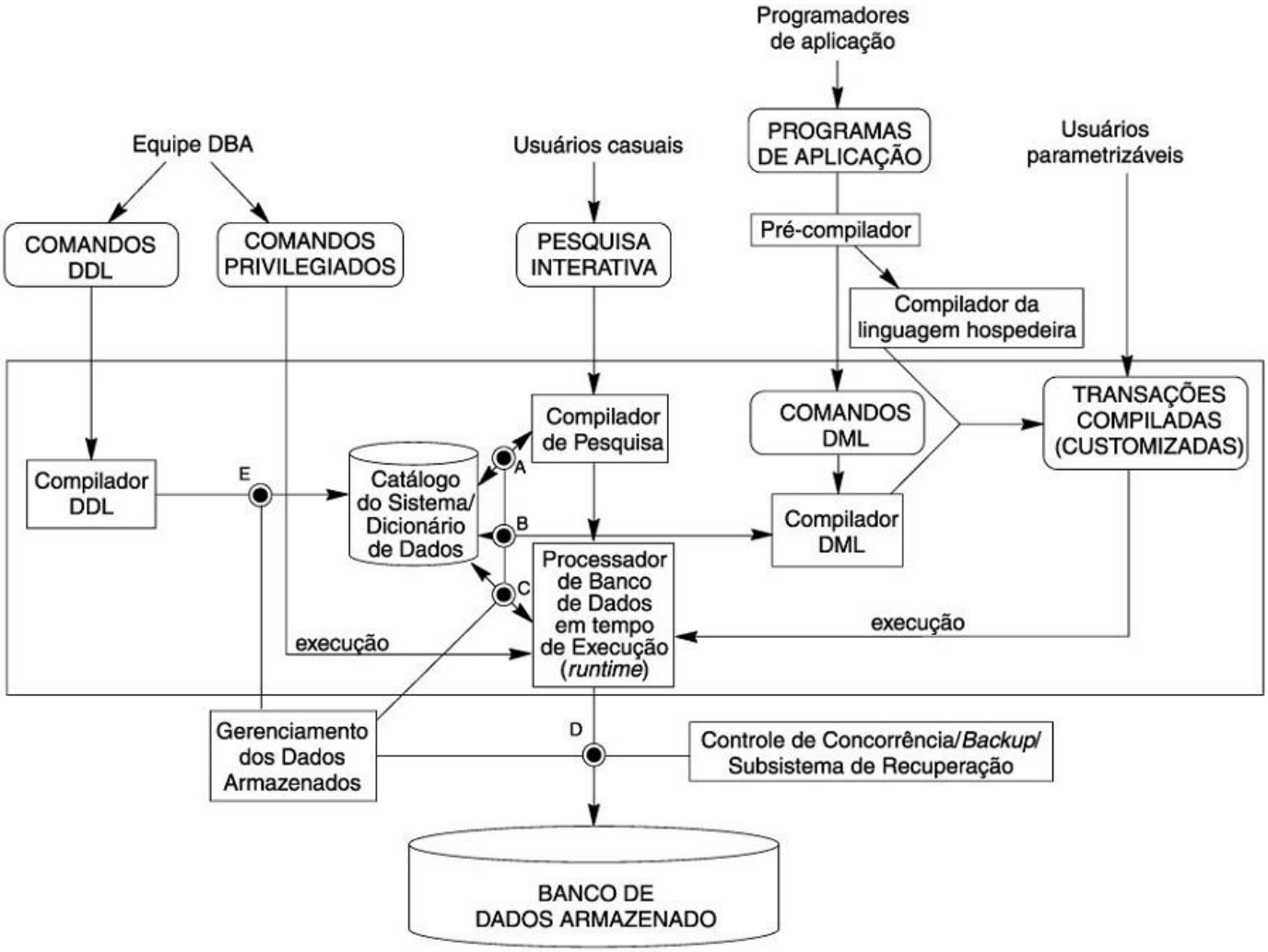
        - Um exemplo de uso é como uma fila (FIFO – First-In-First-Out), onde os elementos são retirados da lista na mesma sequência em que são adicionados.
    - **Vector** *(implements List)*
        - Implementação de List com o mesmo comportamento da *ArrayList*, porém totalmente sincronizada.
        - Por ter seus métodos sincronizados, tem performance inferior ao de uma *ArrayList*, mas pode ser utilizado em ambiente multitarefa (acessado por várias threads) sem perigo de perda da consistência de sua estrutura interna.
        - ...
        - Portanto, anão ser que hajam acessos simultâneos de várias threads à lista, e a nossa sincronização simples, provida pela classe Vector, seja o bastante, Prefira outras implementações como ArrayList ou LinkedList, que oferecem performance superior.
        - **Stack** *(implements List)*
            - Implementação de List que oferece métodos de acesso para uso da lista como uma pilha (LIFO – Last-In-First-Out), como push(), pop() e peek().
            - Estende Vector, portanto herda as vantagens e desvantagens da sincronização deste
            - Pode ser usado para se aproveitar as implementações das operações específicas de pilha.
- **Set** *(interface)*
    - Interface que define uma coleção, ou conjunto que não contém duplicatas de objetos.
    - Isto é, são ignoradas as adições caso o objeto ou um objeto equivalente já exista na coleção.
    - Por objetos equivalentes, entenda-se objetos que tenham o mesmo código hash (retornado pelo método hashCode()) e que retornem verdadeiro na comparação feita pelo método equals().
    - Não é garantida a ordenação dos objetos, isto é, a ordem de interação dos objetos não necessariamente tem qualquer relação com a ordem de inserção dos objetos.
    - Por isso, não é possível indexar os elementos por índices numéricos, como em uma List.
    - **HashSet** *(implements Set)*
        - Implementação se Set que utiliza uma tabela *hash* (a implementação da Sun utiliza a classe HashMap internamente) para guardar seus elementos.
        - Não garante a ordem de interação, nem que a ordem permanecerá constante com o tempo (uma modificação da coleção pode alterar a ordenação geral dos elementos).
        - Por utilizar o algoritmo de tabela hash, o acesso é rápido, tanto para leitura quando para modificação.
    - **LinkedHashSet** *(Implements Set)*
        - Implementação de Set que estende HashSet, mas adiciona previsibilidade à ordem de interação sobre os elementos, isto é, uma interação sobre seus elementos (utilizando o Iterator) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior).
        - Internamente, é mantida uma lista duplamente encadeada que mantem esta ordem.
        - Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leva queda na performance em relação à HashSet, mas ainda é mais rápida que uma TreeSet, que utiliza comparações para determinar a ordem dos elementos.
- **Map** *(Interface)*
    - Interface que define um array associativo, isto é, ao invés de números, objetos são usados como chaves para recuperar os elementos.
    - As chaves não podem se repetir (seguindo o mesmo princípio da interface Set), mas os valores podem ser repetidos para chaves diferentes.
    - Um Map também não possui necessariamente uma ordem definida para o percorrimento.
    - **HashMap** *(implements Map)*
        - Implementação de *Map* que utiliza uma tabela *hash* para armazenar seus elementos.
        - O tempo de acesso aos elementos (leitura e modificação) é constante (muito bom) se a função hash for bem distribuída, isto é, a chance de dois objetos diferentes retornarem ao mesmo valor pelo método hashCode() é pequena.
    - **LinkedHashMap** *(implements Map)*

        - Implementação de *Map* que estende *HashMap*, mas adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, uma iteração sobre seus elementos (utilizando o Iterator) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior).
        - Internamente, é mantida uma lista duplamente encadeada que mantém esta ordem.
        - Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leva queda na performance em relação à *HashMap*, mas ainda é mais rápida que uma *TreeMap*, que utiliza comparações para determinar a ordem dos elementos.
    - **Hashtable** *(implements Map)*
        - Assim como Vector, a *Hashtable* é um legado das primeiras versões do JDK, igualmente sincronizado em cada uma de suas operações.
        - Pelos mesmos motivos da classe Vector, dê preferência as outras implementações, como HaskMap, LinkedHashMap e TreeMap, pelo ganho de performance.

AULA 8 – Introdução à Banco de Dados

Introdução à Banco de Dados

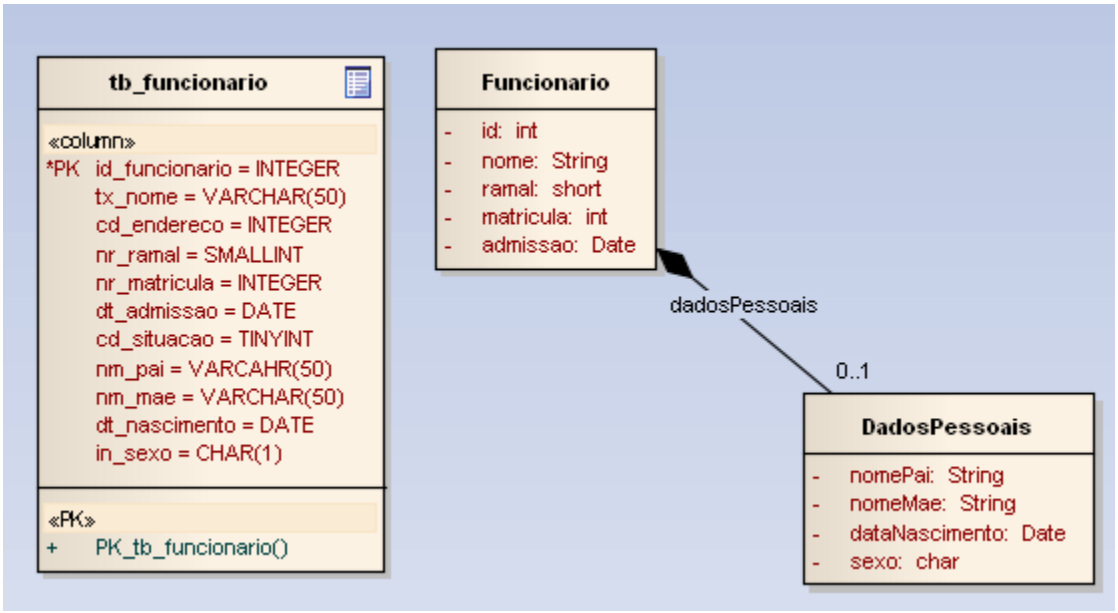
- **O que é um banco de dados?**
  - Banco de dados pode ser definido como um conjunto de "dados" devidamente relacionados;
  - Por "dados" podemos compreender como "fatos conhecidos" que podem ser armazenados e que possuem um sindicado implícito.
  - Um banco de dados possui as seguintes propriedades:
    - É uma **coleção lógica coerente** de dados com um **significado inerente**;
    - Uma disposição **desordenada** dos dados **não pode** ser referenciada como um banco de dados;
- **História dos Sistemas de BD**
  - **Década de 1950 e início da década de 1960:**
    - Processamento de dados usando fitas magnéticas para armazenamento:
      - Fitas fornecem apenas acesso sequencial;
    - Cartões perfurados para entrada.
  - **Final da década de 1960 e década e de 1970:**
    - Discos rígidos permitem acesso direto aos dados;
    - Ted Codd define o modelo de dados relacional:
      - Ganharia o **ACM Turing Award** por este trabalho;
      - IBM Reserach inicia o protótipo do System.
      - UC Berkeley inicia o protóVpo do Ingres.
        - http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf
    - Processamento de transação de alto desempenho (para a época).
  - **Edgar Frank "Ted" Codd: (1923 - 2003)**
    - Recebeu seu PhD em 1965 pela universidade de Michigan;
    - Inventou o modelo relacional enquanto trabalhava na IBM.
  - **Década de 1980:**
    - Protótipo relacionais de pesquisa evoluem para sistemas comerciais;
      - SQL se torna o padrão do setor;
    - Sistemas de banco de dados paralelos e distribuídos;
    - Sistemas de banco de dados orientados a objeto.
  - **Década de 1990:**
    - Grandes aplicações de suporte a decisão e exploração de dados;
    - Grandes data warehouses de vários terabytes;
    - Surgimento do comércio Web.
  - **Década de 2000:**
    - Padrões XML e Xquery;
    - Administração de banco de dados automaVzados.
- **Componentes de um SGBD**



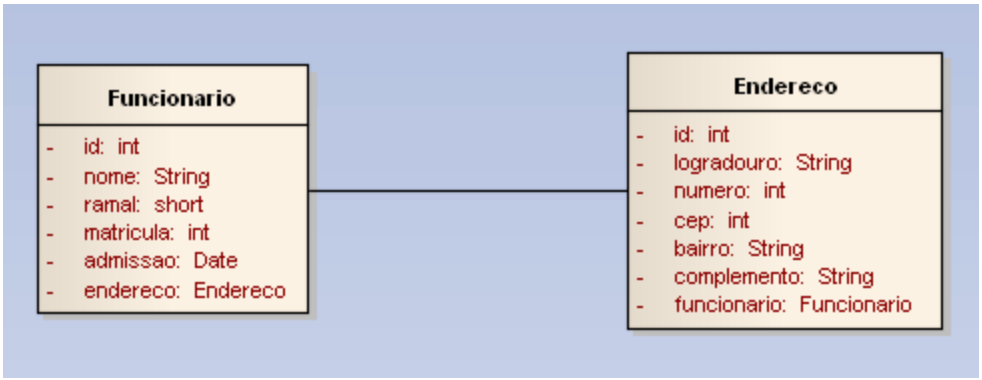
Entidade e Relacionamento

- **Entidades**
  - São classes, cujos objetos representam elementos existentes no **mundo real** (nomes, pessoas, valores, documentos, etc.), em torno dos quais o software é desenvolvido.
  - Também são conhecidas como classes de domínio da aplicação.
  - Um objeto pode ser considerado persistente quando seus atributos podem ser armazenados ao término de uma sessão de utilização do software e recuperado ao início de uma outra sessão.
  - Existem diversas alternativas para persistir objetos
    - **Banco de dados relacional**;
    - Banco de dados orientado a objetos;
    - Serialização de objetos em sistemas de arquivos;
    - Documentos XML.
  - Diante do repositório de dados, um objeto estará sujeito a quatro operações de persistência (CRUD):
    - Criação, inserção ou inclusão no repositório (Create);
    - Leitura ou recuperação dos dados (Read);
    - Atualização ou modificação dos dados (Update);
    - Remoção ou eliminação dos dados (Delete).
- **Relacionamentos**
  - No desenvolvimento de software de complexidade média ou elevada, os modelos de dados recorrem aos **relacionamentos entre entidades** para expressar **situações do mundo real**;
  - A UML define **associações** como sendo o **relacionamento entre os objetos**, em tempo de execução.
  - Nos modelos de dados relacionais as associações de objetos encontram suporte nas **chaves estrangeiras** estabelecidas entre tabelas;
  - A cardinalidade está diretamente ligada ao tipo de relacionamento entre as entidades.
  - **Composição <u>Umpara-Um</u>**

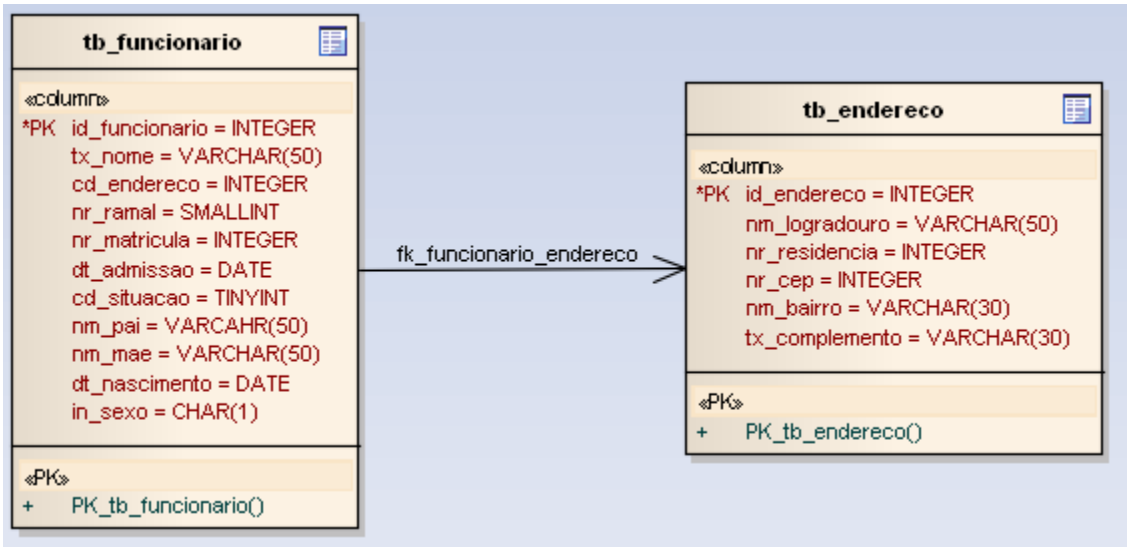
- Em determinadas situações a granularidade do modelo de classes pode ser maior do que a granularidade do modelo de banco de dados.
    - Por exemplo, duas classes que se relacionam na forma um para um podem ser mapeadas para uma única tabela no banco de dados.
- Exemplo:
    - Vamos considerar que cada funcionário possui um conjunto de informações modeladas como seus dados pessoais.
    - Tal conjunto de informação existe em função de um funcionário, e é intransferível. Isto caracteriza um relacionamento de composição, com cardinalidade um para um.
    - A composição em orientação a objetos define uma associação forte, onde as partes estarão sempre associadas ao mesmo objeto todo (classe marcada com o losango preto).



- **Associação Um para Um**
    - Mapeamento Um-para-Um e Navegação Bidirecional:
        - O primeiro caso de associação a ser estudado é a associação bidirecional entre Funcionario e Endereco:

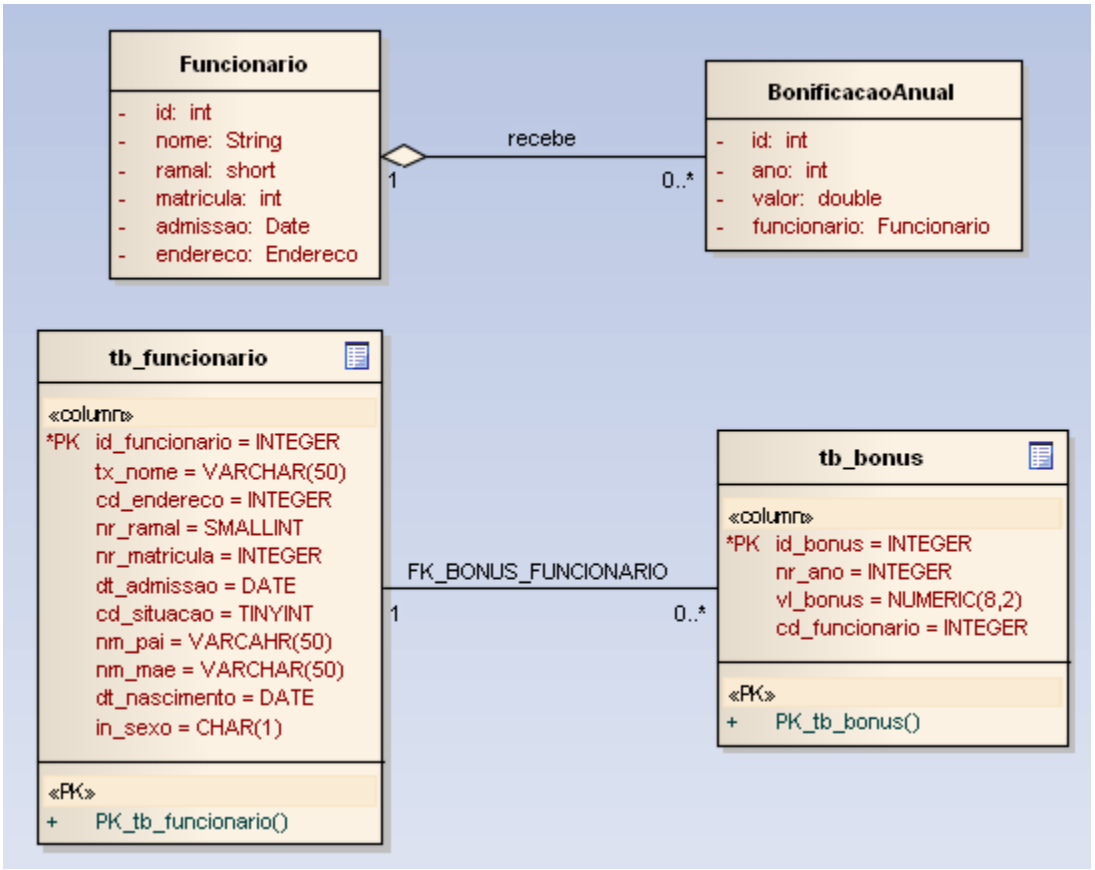


- Nos bancos de dados relacionais, é possível estabelecer o seguinte modelo de tabelas:



- A coluna `cd_endereco` de `tb_funcionario` é a coluna que suporta o relacionamento.
- Consideramos então os funcionários como os "proprietários" do relacionamento.
- Observe que, na coluna `cd_endereco`, há uma chave estrangeira (***foreign key***) está estabelecendo o caminho unidirecional "do funcionario para o endereço".

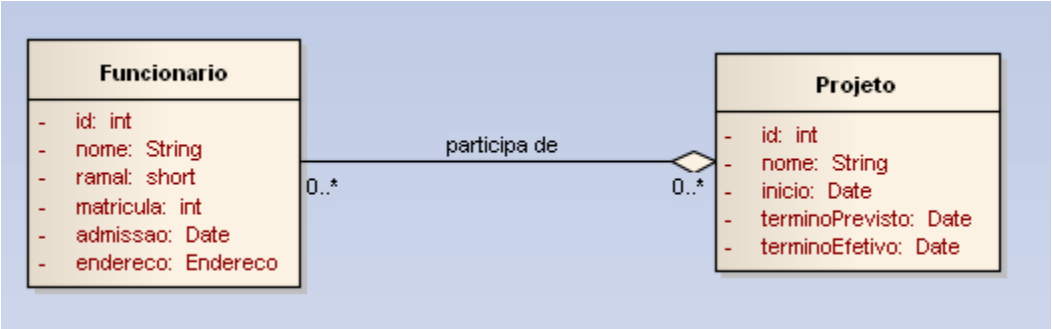
- **Associação <u>Um para Muitos</u>**
  - Frequentemente os modelos de dados apresentam relacionamentos do tipo "um-para-muitos".
  - Será considerado o caso "Um funcionário recebe muitas bonificações";
  - Inicialmente vamos considerar a navegação bidirecional.



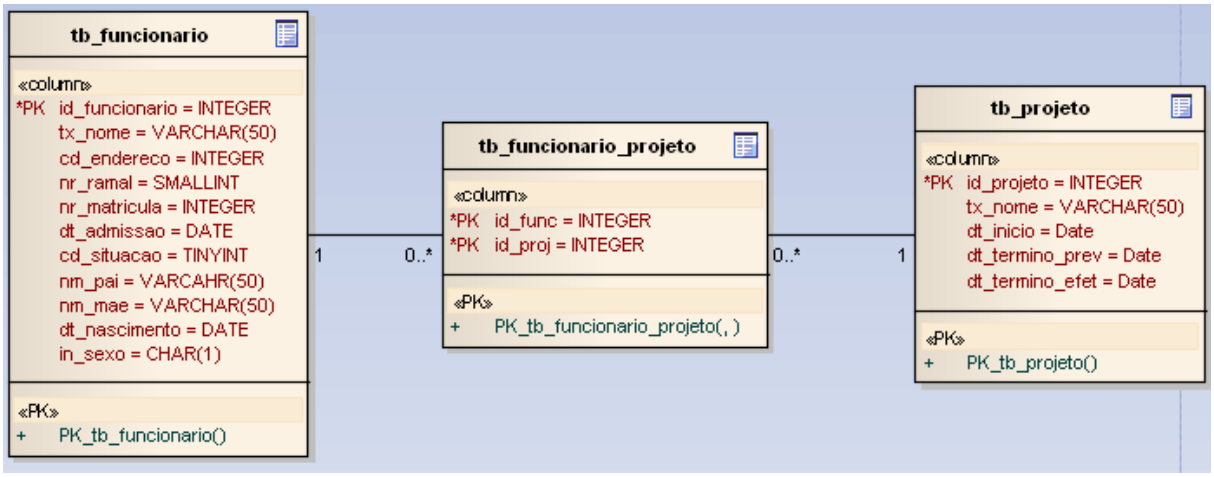
  - No banco de dados é necessário uma coluna de que suporta o relaciona na tabela "muitos". Neste caso, trata-se da coluna `cd_funcionario` da tabela `tb_bonus`. É desejável a presença de uma chave estrangeira nesta coluna.
  - ***Em Java existe várias formas de implementar este tipo de associação, com o auxílio de conjuntos, listas e mapas;***

- **Associação <u>Muitos-para-Muitos</u>**
  - É o caso onde as duas extremidades de uma associação têm multiplicidade "muitos".

- No modelo relacional surge uma tabela intermediária para associar muitos funcionários a muitos projetos:



- No modelo relacional surge uma tabela intermediária para associar muitos funcionários a muitos projetos:



SQL

- **Tipos de Dados:**

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision, exact	no limit
numeric	variable	user-specified precision, exact	no limit
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

- **Criação e remoção de tabelas**
  - **SQL-DDL (Criando tabelas)**
    - Define-se uma relação SQL usando o comando: ***CREATE TABLE***.

```
CREATE TABLE nome_tabela (A1 D1, A2 D2, <regra de integridade>);
```

    - cada ***Ai*** é o nome de um atributo no esquema da tabela;
    - ***Di*** é o tipo de domínio (tipo do dado) dos valores do atributo;
  - As principais regras de integridade são:

    `primary key(Aj1, Aj2, ..., Ajm)`:
    - A especificação primary key diz que os atributos: Aj1, Aj2, ..., Ajm formam a chave primária da relação (tabela);
    - É necessário que a chave primaria seja não nula e única!
  - As principais regras de integridade são:

`foreign key(Aj1, Aj2, ..., Ajm) references r`:

- A cláusula foreign key inclui a relação dos atributos que constituem a chave estrangeira (Aj1, Aj2, ..., Ajm);
- É utilizando ***r*** quanto ao nome da relação à qual a chave estrangeira faz referência.
- EXEMPLO:

```
CREATE TABLE TESTE (
    COD_TESTE INTEGER PRIMARY KEY,
    NOME_TESTE VARCHAR
);


CREATE TABLE TESTE2 (
    COD_TESTE2 INTEGER PRIMARY KEY,
    COD_TESTE INTEGER REFERENCES TESTE,
    NOME_TESTE2 VARCHAR
);

CREATE TABLE TESTE3(
    COD_TESTE INTEGER,
    COD_TESTE2 INTEGER,
    NOME_TESTE3 VARCHAR,
    FOREIGN KEY (COD_TESTE) REFERENCES TESTE,
    FOREIGN KEY (COD_TESTE2) REFERENCES TESTE2,
    PRIMARY KEY (COD_TESTE, COD_TESTE2)
);
```

- **SQL-DDL (Remover Tabelas)**
- O comando **DROP TABLE** remove todas as informações de uma relação do banco de dados.
- Tanto os dados como o esquema é excluído.

```
DROP TABLE <nome_tabela>;
```

- Onde <nome_tabela> é a tabela que se deseja excluir do banco de dados.

- **Manipulando os dados de um banco de dados**
  - **Remoção**
    - Um pedido para remoção de dados é expresso muitas vezes do mesmo modo que uma consulta;
    - Pode-se remover somente tuplas inteiras;
    - Em SQL a remoção é expressa por:

```
DELETE FROM tabela WHERE predicado;
```

    - Exemplo:
      - Remover todos os empréstimos com total entre 1300 e 1500 reais:
        `DELETE FROM emprestimo WHERE total BETWEEN 1300 AND 1500;`
  - **Inserção**
    - Para inserir dados em uma tabela podemos especificar uma tupla a ser inserida ou escrever uma consulta cujo o resultado é a um conjunto de tuplas a serem inseridas;
    - A inserção é expressa por:

```
INSERT INTO tabela(A1, A2) VALUES (V1, V2);
```

    - Onde Ai representa os atributos a serem inseridos e Vi os valores.
    - Exemplo:
      - Deve-se inserir a informação que existe um conta número 25 na agencia “Agencia13” e que ela tem um saldo de 199 reais:

```
INSERT INTO conta (numero_conta, nome_agencia, saldo) VALUES (25,
                                'Agencia13', 199);
```

    - A instrução ***insert*** pode conter subconsultas que definem os dados a serem inseridos:

```
INSERT INTO tabela1 (A1, A2) SELECT (A1, A2) FROM tabela2;
```

    - Com isto a instrução **insert** insere mais de uma tupla.

- **Atualizações**
- Em determinadas situações, pode-se querer modificar valores das tuplas.
- Para alterações utiliza-se o comando **UPDATE**.

```
UPDATE tabela SET A1 = V1, A2 = V2 WHERE predicado;
```

- Onde **Ai** representa o atributo em questão e **Vi** o valor alterado.
- Exemplo:
  - Suponha que o pagamento da taxa de juros anual esteja sendo efetuado e todos os saldos deverão ser atualizados de em 5%:

```
UPDATE conta SET saldo = saldo * 1,05;
```

- Exemplo:
  - Suponha que agora contas com saldo superior a 10000 reias, recebam 6 por cento de juros:

```
UPDATE conta SET saldo = saldo * 1,06
    WHERE saldo > 10000;
```

- **Campo Serial**
  - O campo serial é uma alternativa para a criação de campos auto-incrementáveis:

```
CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT, age INT4);
```

  - Para inserir valores na tabela criada no slide anterior:

```
INSERT INTO users (name, age) VALUES ('Mozart', 20);
```

  - OU

```
INSERT INTO users (name, age, id) VALUES ('Mozart', 20, DEFAULT);
```

- **Recuperando dados**
  - **Estrutura Básica**
    - Uma consulta }pica em SQL tem a seguinte forma:

```
SELECT a1, a2, a3, ..., an FROM r1, r2, ..., rn WHERE P;
```

      - Cada ai representa um atributo de cada ri (relação), P é o predicado (condição).
      - A consulta é equivalente a seguinte expressão em álgebra relacional:

  - **Cláusula SELECT**
    - Exemplo: Encontre os nomes de todas as agencias da relação empréstimo:

```
SELECT nome_agencia FROM emprestimo;
```

    - No caso em que desejamos a forçar a eliminação, podemos inserir a palavra-chave **DISTINCT** depois do **SELECT**.

```
SELECT DISTINCT nome_agencia FROM emprestimo;
```

    - O asterisco '*****' pode ser usado para denotar "todos os atributos", assim para exibir todos os atributos de empréstimo:

```
SELECT * FROM emprestimo;
```

    - SELECT pode conter expressões aritméticas envolvendo operadores: +,--,/ e *:

```
SELECT nome_agencia, total * 100 FROM emprestimo;
```

  - **Cláusula WHERE**
    - Considere a consulta:
      - "Encontre todos os números de empréstimos feitos na agência "Central" com totais de emprésVmos acima de 1200 dólares":
      - Esta consulta pode ser escrita em SQL como:

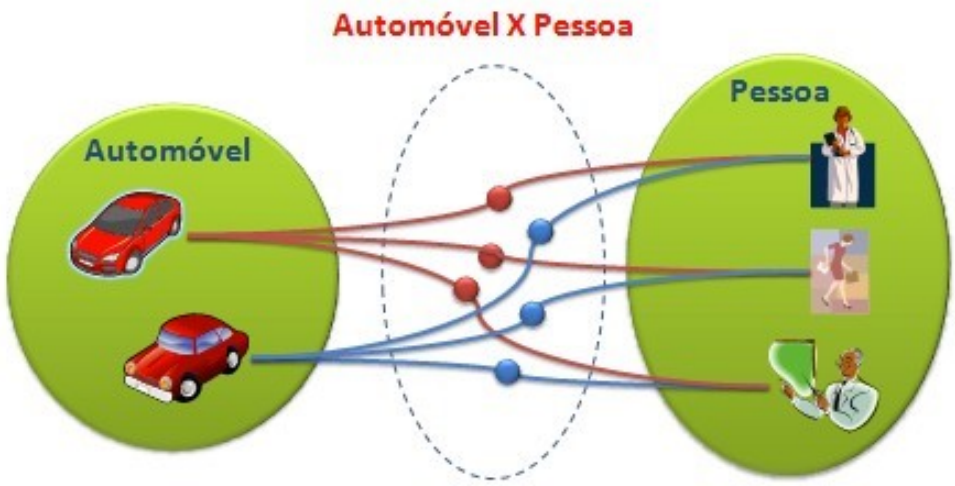
```
SELECT numero_emprestimo  FROM emprestimo
WHERE nome_agencia = 'Central' AND total > 1200;
```

    - SQL usa conectores lógicos **and**, **or** e **not** na cláusula **where**;

- Os operadores dos conectivos lógicos podem ser expressões envolvendo operadores de comparação: **<**, **<=**, **>**, **>=** e **<>**
- SQL possui operador de comparação **`between`** para simplificar a cláusula **`where`** que especifica valores dentro de um intervalo.
    - Se desejarmos saber os números de emprésVmos cujo o montante esteja entre 90000 e 100000, pode-se fazer:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total BETWEEN 90000 AND 100000;
```

- **Cláusula FROM**
    - A cláusula **from** por sí só define um produto cartesiano das relações da cláusula.



- Expressão sem produto cartesiano:

$$(\sigma \quad (devedor \times emprestimo)))$$

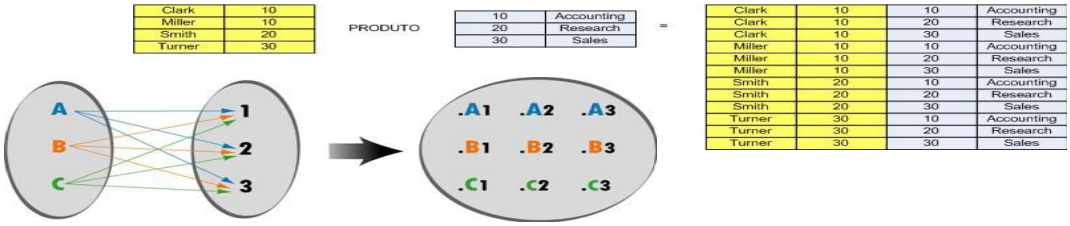
$$(\pi \quad devedor.numero\_emprestimo = emprestimo.numero\_emprestimo$$

$$nome\_cliente,\ numero\_emprestimo$$

- Para consultar todos os clientes que tenham um empréstimo em um banco, encontre seus nomes é números de empréstimo, em SQL pode ser escrito:

```
SELECT DISTINCT nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

- **Importante**: na consulta anterior usa-se a notação **`nome_relação.nome_atributo`**, como na álgebra relaciona para evitar ambiguidade nos casos em que um atributo apareça em mais de uma relação.
- Para impedir a formação de um produto cartesiano, sempre inclua uma condição de restrição válida na cláusula **WHERE**.
    - Relembrando produto cartesiano:



- Comparação
    - Com produto cartesiano:

Clark	10
Miller	10
Smith	20
Turner	30

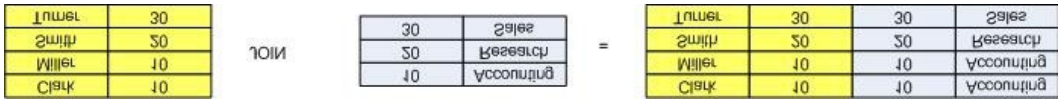
PRODUTO

10	Accounting
20	Research
30	Sales

=

Clark	10	10	Accounting
Clark	10	20	Research
Clark	10	30	Sales
Miller	10	10	Accounting
Miller	10	20	Research
Miller	10	30	Sales
Smith	20	10	Accounting
Smith	20	20	Research
Smith	20	30	Sales
Turner	30	10	Accounting
Turner	30	20	Research
Turner	30	30	Sales

- Com Junção:



- **Variáveis Tuplas**
  - Variáveis tuplas são definidas na cláusula **FROM** por meio do uso da cláusula **AS**.
  - A seguir é apresentada a consulta "para todos os funcionários encontre o nome do seu supervisor":

```
SELECT DISTINCT f.nome_funcionario, s.nome_funcionario
    FROM funcionario AS f, funcionario AS s
WHERE f.rg_funcionario = s.rg_funcionario;
```

- **Operações em String**
  - As operações em Strings mais usadas são as checagens para verificação de coincidências de pares, usando o operador like.
  - Indicaremos esses pares por meio do uso de caracteres especiais:
    - Porcentagem (**%**): compara qualquer substring;
    - Sublinhado (**_**): compara qualquer caracter.
  - Comparações desse tipo são sensíveis ao tamanho das letras; isto é, minúsculas não são iguais a maiúsculas e vice--versa;
  - Para ilustrar considere os seguintes exemplos:
    - 'Pedro%' corresponde a qualquer String que comece com Pedro.
    - '%inh%' corresponde a qualquer String que possua uma substring 'inh', por exemplo: 'huguinho', 'zezinho' e 'luizinho'.
  - Pares são expressões em SQL usando o operador de comparação LIKE. Considere a consulta: "encontre os nomes de todos os clientes que possuam 'Silva' ":

```
SELECT nome_cliente FROM cliente
WHERE nome_cliente LIKE '%Silva%';
```

  - SQL permite pesquisar diferenças em vez de coincidências usando **NOT LIKE**.

- **Ordenação e Apresentação de Tuplas**
  - SQL oferece ao usuário algum controle sobre a ordenação por meio da qual as tuplas de uma relação serão apresentadas.
  - A cláusula **ORDER BY** faz com que as tuplas do resultado de uma consulta apareçam em uma determinada ordem.
  - Para listar em ordem alfabética todos os clientes que tenham um empréstimo na agência Centro:

```
SELECT DISTINCT nome_cliente
    FROM devedor AS d, emprestimo AS e
WHERE d.numero_emprestimo = e.numero_emprestimo
    AND nome_agencia = 'Centro'
    ORDER BY nome_cliente;
```

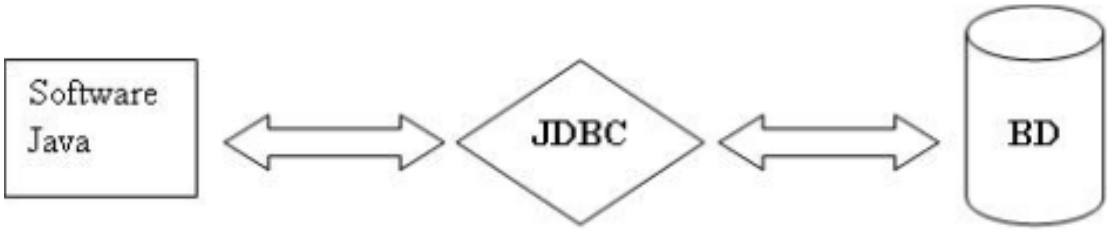
  - Por default a cláusula **ORDER BY**, relaciona os itens em ordem acedente (**ASC**), para inverter a ordem pode-se utilizar **DESC**.

```
SELECT cidade_cliente, nome_cliente
FROM cliente
ORDER BY cidade_cliente ASC, nome_cliente DESC;
```

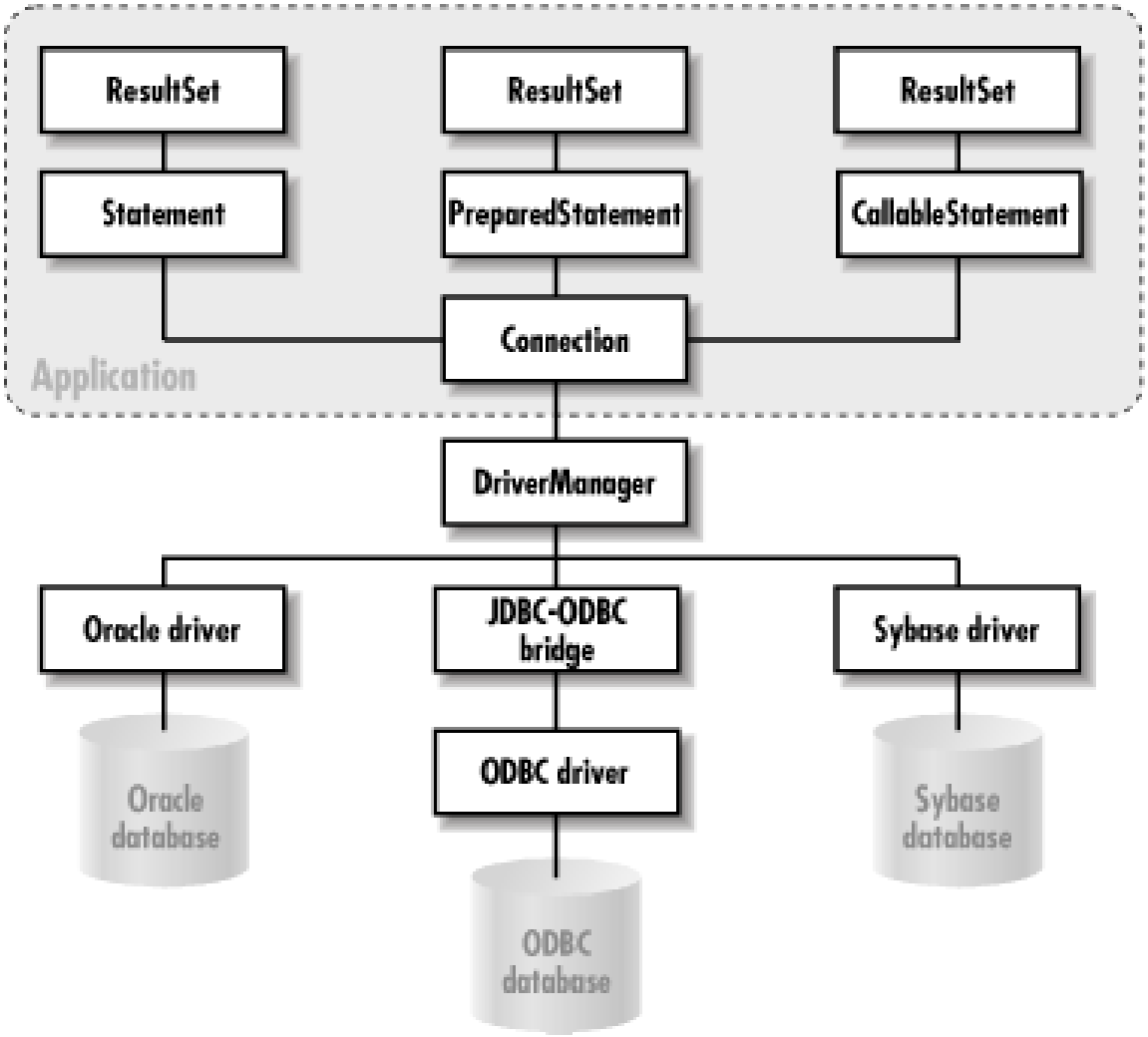
- Exercícios Modelo base do Exercício

Mas como eu vou fazer o Java se comunicar com o SGBD?

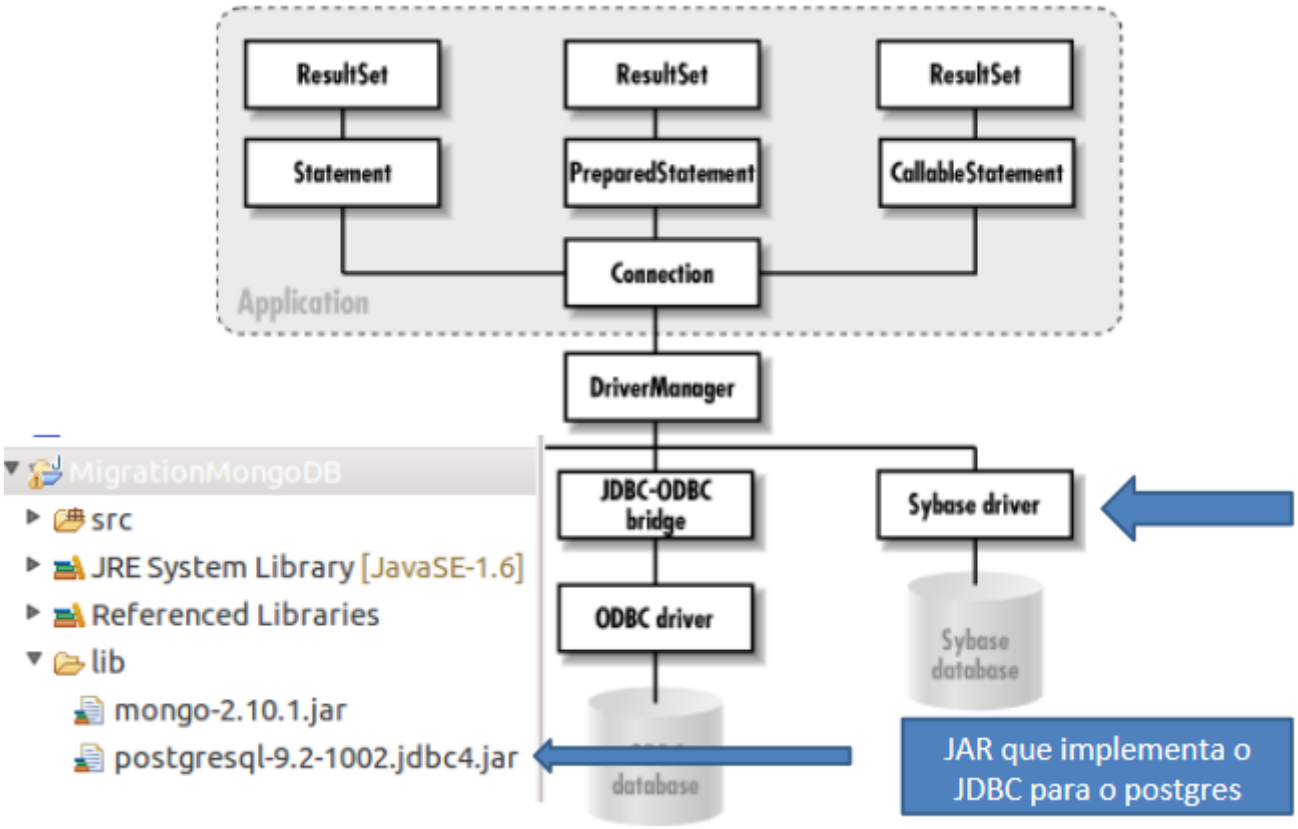
- **JDBC**
  - O mecanismo existente na linguagem Java que permite tal conexão com bancos de dados corresponde a um pacote de classes chamado de JDBC (Java Data Base ConnecTon).
  - Abaixo é mostrada uma figura que tenta ilustrar como o JDBC funciona: JDBC



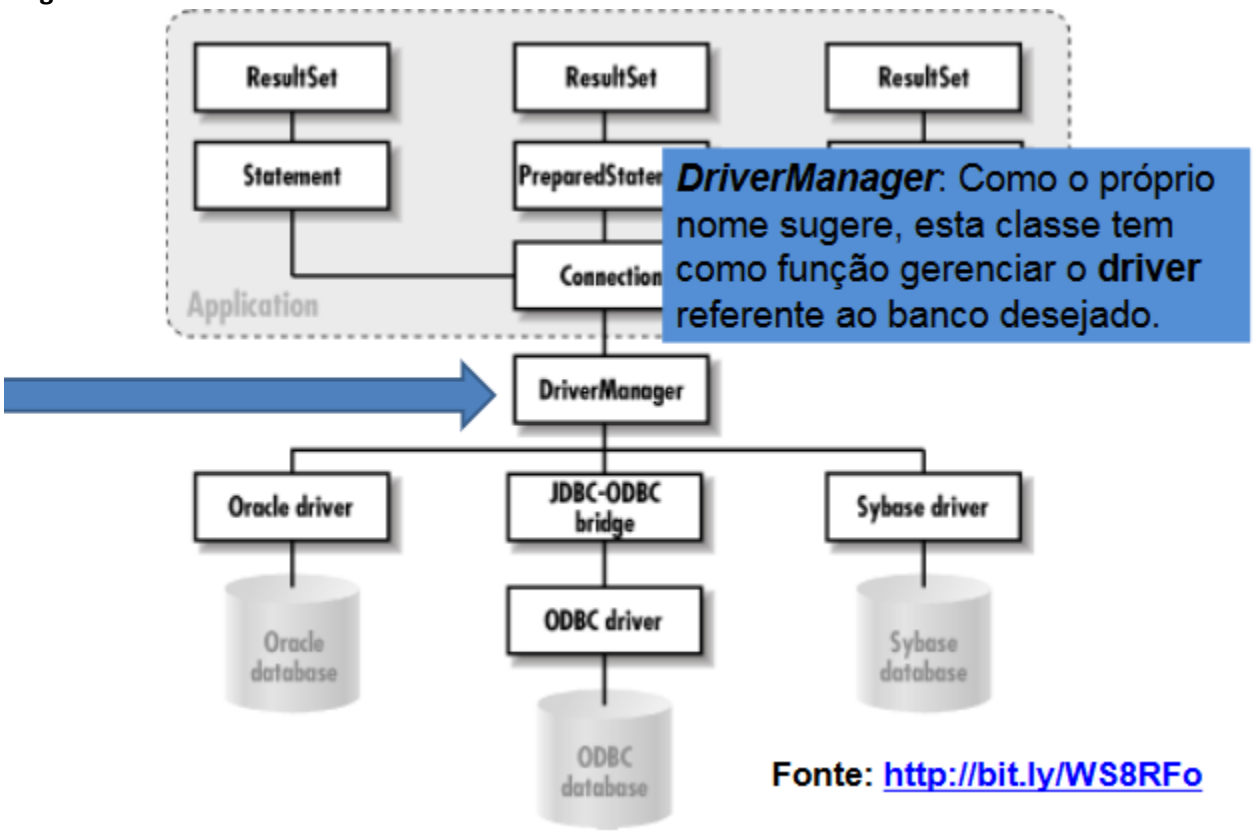
  - Os comandos usados para estabelecer a comunicação com os bancos de dados, independente de qual banco seja usado:
    - Postgres
    - Oracle
    - Sybase
    - MySQL
    - entre outros
  - Serão sempre da mesma forma.
  - Tudo o que é necessário é que o software que você está desenvolvendo utilize as classes e métodos definidos no pacote JDBC (.jar);
  - Uma grande vantagem na utilização deste pacote, é conseguir uma independência entre o aplicativo Java e a plataforma de banco de dados utilizada.
- **Visão Geral**



- **JDBC**



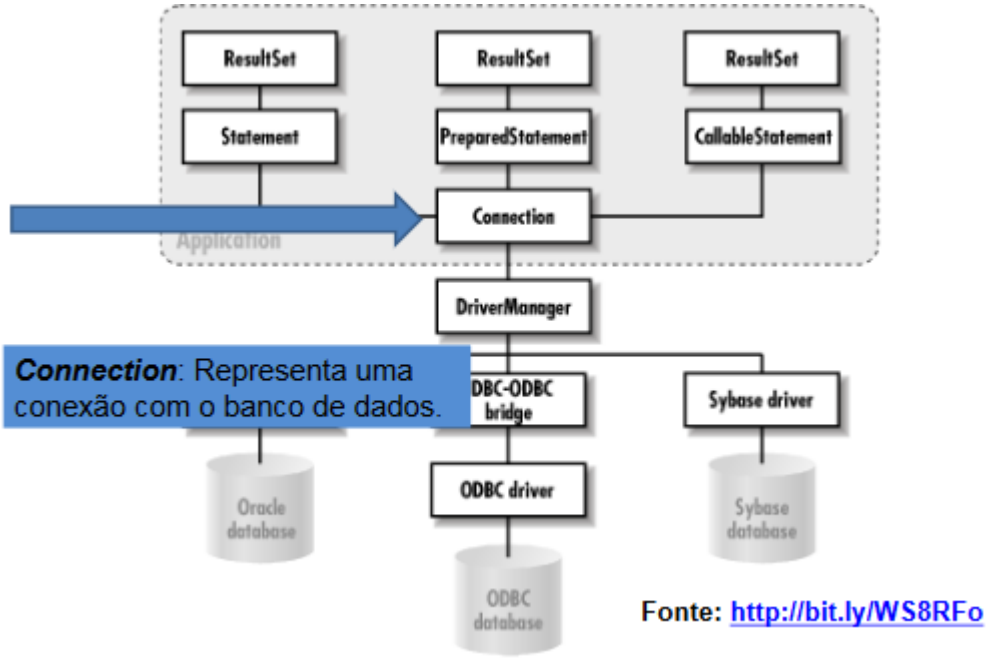
- **DriverMnager**



  - **Principais Métodos**

static Connection	**getConnection**(String url) Attempts to establish a connection to the given database URL.
static Connection	**getConnection**(String url, Properties info) Attempts to establish a connection to the given database URL.
static Connection	**getConnection**(String url, String user, String password) Attempts to establish a connection to the given database URL.

- **Connection**



Fonte: http://bit.ly/WS8RFo

- **Exemplos da Criação de Conexão**

```
import java.sql.Connection;
import java.sql.DriverManager;

public class DatabaseService {

    public static Connection getConnPostgres() {
        Connection conn = null;

        try {
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.0.247:5438/database",
                    "user", "password");
        } catch (Exception e) {
            System.err.println(e);
        }

        return conn;
    }
}
```

- **Exemplo do uso da conexão**
  - *Imagem*

```java
public void metodoBD() {
    Connection conn = null;

    try {
        conn = DatabaseService.getConnPostgres();

        //TODO fazer a operação desejada

    } catch (Exception e) {
        System.err.println(e);

    } finally {
        if(conn != null) {
            try {
                conn.close();
            } catch (Exception e) {}
        }
    }
}
```

O método abriu recurso? Então depois do seu uso deve-se fechá-lo

- **Statement's**
  - *Imagem*
  - Mas quando usar cada uma das interfaces?
    - **Statement**: ÚTl quando você estiver usando instruções SQL estáticas em tempo de execução. Esta interface não aceita parâmetros.
    - **PreparedStatement**: Esta interface aceita a inserção de parâmetros em tempo de execução.
    - **CallableStatement**: Deve--se utiliza esta interface quando existe a necessidade de executar uma StoredProcedure
  - **Statement**

```java
public void removerClientes() {
    Connection conn = null;
    Statement st = null;

    try {
        conn = DatabaseService.getConnPostgres();
        st = conn.createStatement();
        st.execute("DELETE FROM CLIENTE");

    } catch (Exception e) {
        System.err.println(e);
    } finally {
        try {
            if(st != null) {
                st.close();
            }
            if(conn != null) {
                conn.close();
            }
        }catch (Exception e) {}
    }
}
```

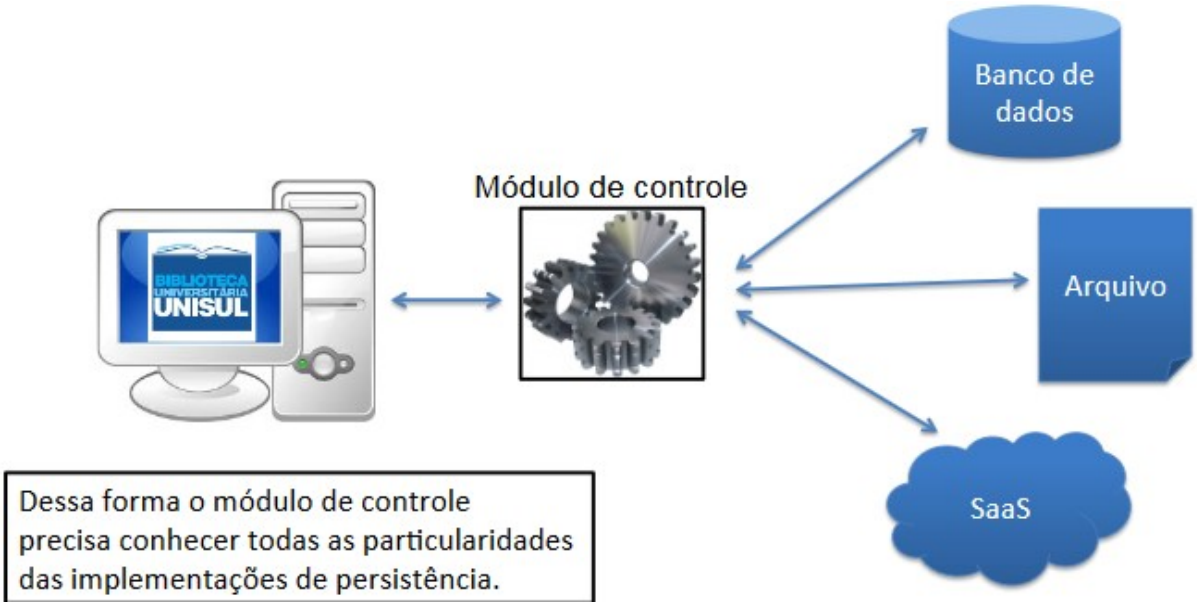
    - *Imagem – código removeClienteParametro(int idCliente)*
    - **Statement – Injeção de SQL**
      - *Imagem – código removeClienteParametro(String nomeCliente)*
  - **PreparedStatement**

```java
public void removeClienteParametroPS(int idCliente) {
    Connection conn = null;
    PreparedStatement ps = null;

    try {
        conn = DatabaseService.getConnPostgres();
        ps = conn.prepareStatement("DELETE FROM CLIENTE WHERE id_cliente = ?");
        ps.setInt(1, idCliente);

        ps.execute();
    } catch (Exception e) {
        System.err.println(e);
    } finally {
        try {
            if(ps != null) {
                ps.close();
            }
            if(conn != null) {
                conn.close();
            }
        }catch (Exception e) {}
    }
}
```

        - Os dados passados como parâmetro são validade e tratados antes da submissão da instrução para o banco de dados;
        - Converte o Tipo do dado para o mais próximo do SGBD específico;
        - Não permite a injeção de SQL.
    - **CallableStatement**
        - *Imagem – código executarStoredProcedure()*
- **ResultSet**
    - *Imagem – código Infográfico*
    - *Imagem – código listarCliente(String estado)*
    -

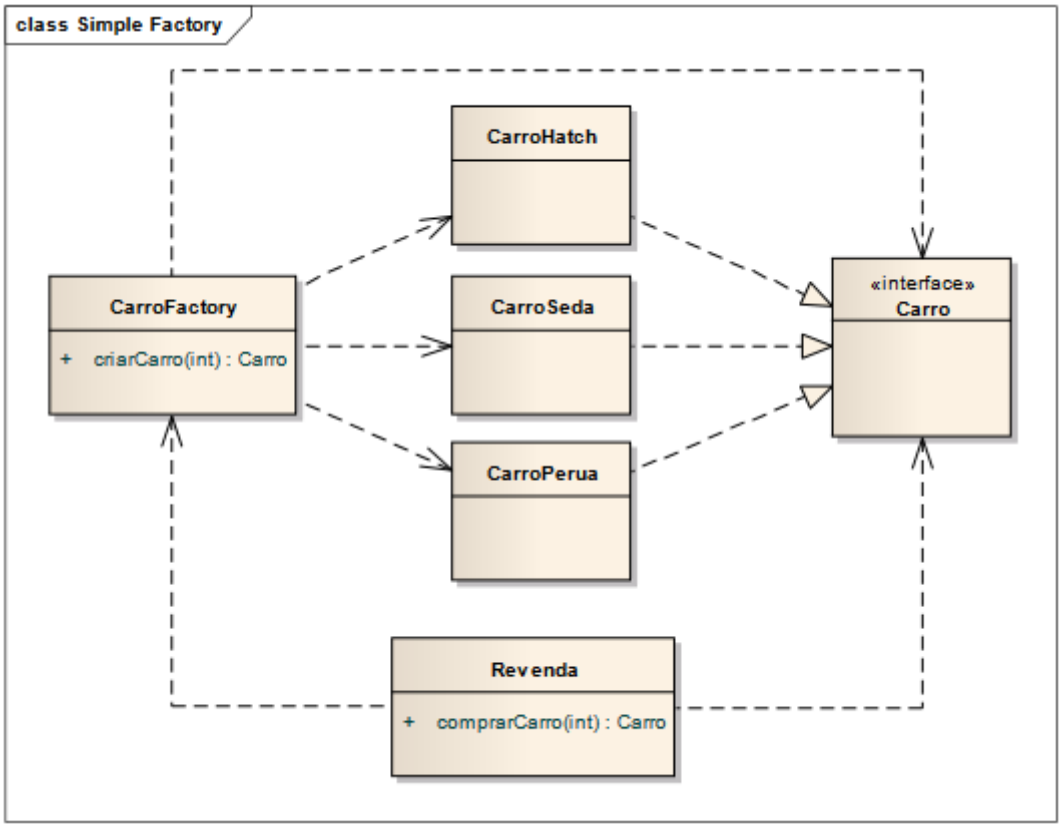
AULA 10 – Padrão de Projeto – Dao Factory

- O que é um padrão de projeto?
    - É um padrão que descreve um problema recorrente em determinado cenário.
    - Não somente descreve o problema em determinados cenários como também descreve a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações.
    - Padrões de projeto são soluções elegantes e reutilizáveis para problemas recorrentes que encontramos diariamente no processo de desenvolvimento de aplicativos para o mundo real.
- Motivação
    - Como tratar os vários tipos de fonte de dados de um projeto de modo que essa implementação seja transparente para o sistema?
    - Vamos pensar no caso da biblioteca universitária que desenvolvemos.
    - Sistema da Biblioteca Universitária da Unisul

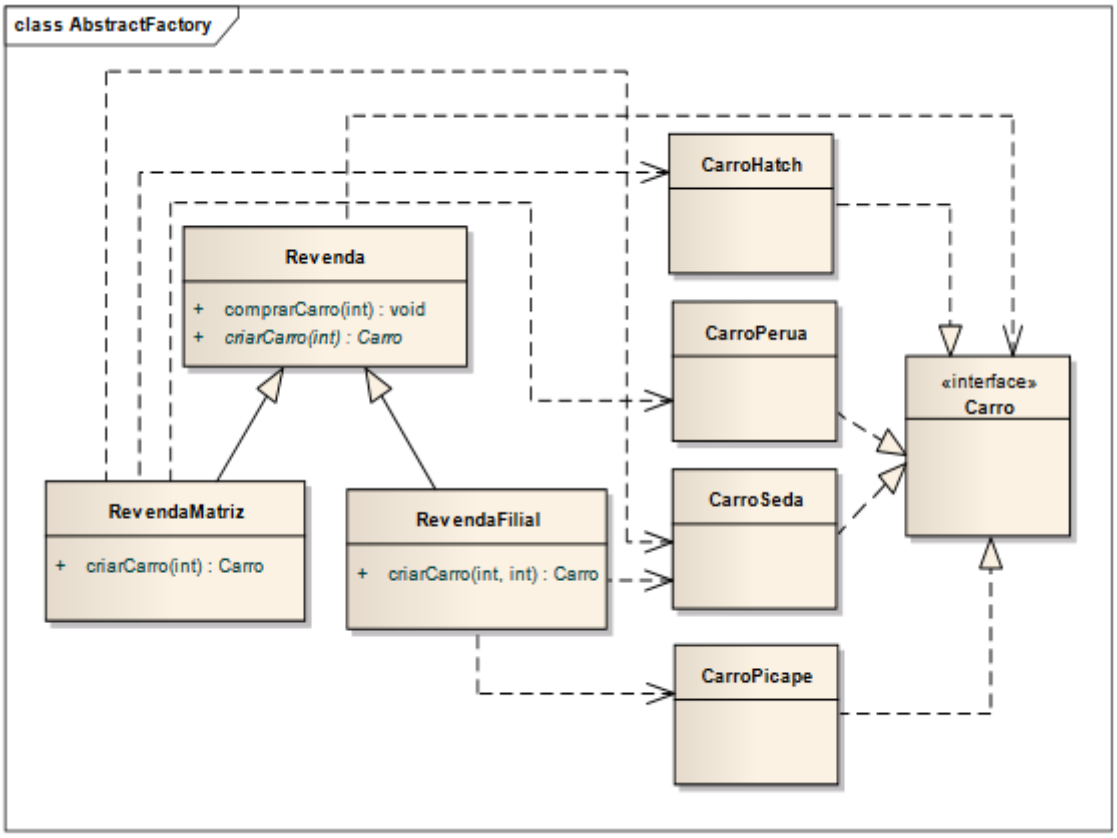


Dessa forma o módulo de controle precisa conhecer todas as particularidades das implementações de persistência.

- **DAO Factory**
  - **DAO (Data Access Object):**
    - Padrão de projeto que tem como objetivo criar interfaces para acesso único aos dados, de modo que a implementação da persistência não tenha dependência direta com a camada de negócio.
  - **Factory:**
    - “Fabrica” de objetos que encapsula a lógica de construção do mesmo.
    - Simple Factory



    - Abstract Factory DAO



- **DAO - Data Access Object**
  - A utilização de um padrão de projeto como o DAO promove o isolamento entre regras de negócios e as regras de persistência de dados;
  - Ou seja, ele promove o isolamento entre classes com objetivos distintos (persistência/negócio/interface);
  - Com isso, ele permite uma maior flexibilidade quando se deseja, por exemplo, utilizar diferentes SGBDs (Sistema Gerenciador de Banco de Dados).
  - Com ele todas as regras de persistência passam a ser mediadas por um objeto do tipo DAO;
  - Toda a lógica de acesso e execução ao banco de dados é colocada dentro do objeto DAO;
  - As operações de CRUD passam então a ser de inteira responsabilidade do objeto DAO, isolando-as do resto da aplicação.