**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

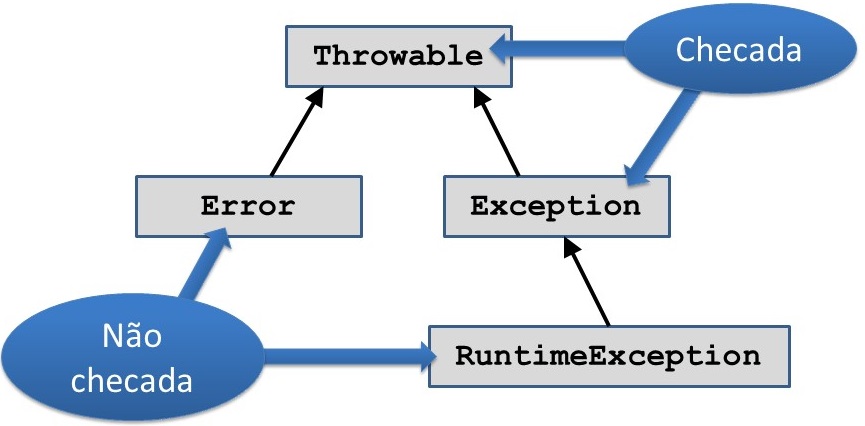
**SLIDE 4 – Collections + Tratamento de exceções**

**Igualdade em Hash**

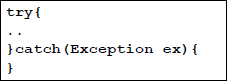
* Para entendermos como é possível identificar se dois elementos (objetos complexos) são iguais e devem ser ou não adicionados em um Hash, formulou-se a seguinte classe:
  + 
* Para adicionar um cliente (ou qualquer outro tipo de objeto complexo) em um hash, é necessário implementar dois métodos:
  + 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.



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

**SLIDE 5 – Manipulação de Arquivos**

Manipulação de arquivos

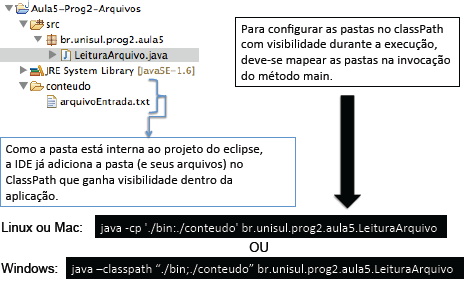
* Diariamente utilizamos arquivos em nossos computadores, para armazenar e consultadas dados.
* Este é um recurso bastante importante em programação, pois permite que dados sejam persistidos ao final da execução de um programa.
* Até então mantínhamos os dados de nossas aplicações em memória, de modo 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.



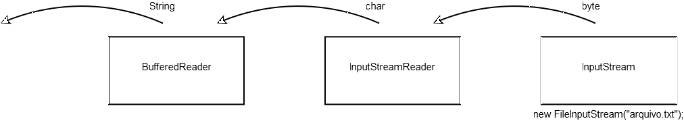
* **Classe File**
  + Essa classe está diretamente ligada à um caminho;
  + Esse caminho pode existir ou não;
  + Através deste objeto é possível verificar propriedades de um arquivo ou pasta.
    - Mas ele não interage com o seu conteúdo.
  + É possível também criar arquivos e pastas, mas não escrever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



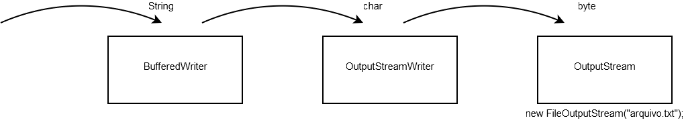
* Criando subpasta
* Criando arquivo

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**

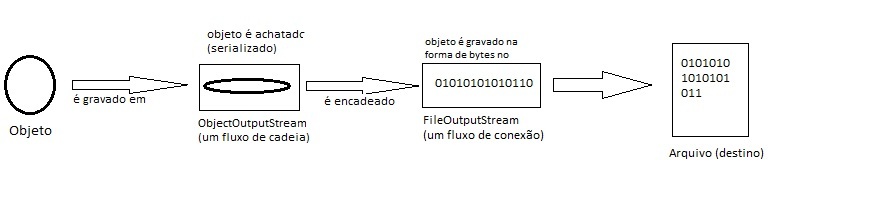


* + É possível escrever apenas no final do arquivo, sem sobrescrevê-lo?
    - Sim. Basta adicionar o parâmetro true, na construção do objeto da classe FileOutputStream.

**SLIDE 5 – Serialização e a manipulação de arquivos**

Serialização

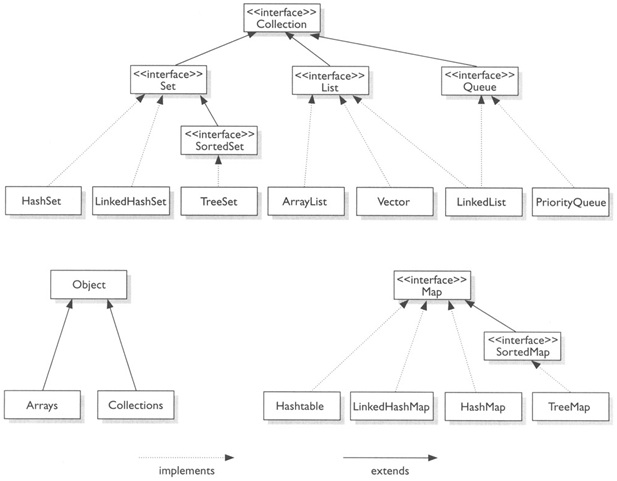
* **O que é serialização?** 
  + A serialização resume-se em um único objetivo: salvar, gravar e capturar o estado de um objeto.
  + Ou seja, tenho um objeto de uma classe e quero salvar seu estado, então uso serialização, pois quando quiser usar esse objeto com seu último estado gravado é somente deserializá­lo.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