E eles serão meus, diz o Senhor dos Exércitos; naquele dia serão para mim jóias; poupá-los-ei, como um homem poupa a seu filho, que o serve.

Então voltareis e vereis a diferença entre o justo e o ímpio; entre o que serve a Deus, e o que não o serve.

Malaquias 3: 17-18

Curso de Especialização em Tecnologia Java

LINGUAGEM DE PROGRAMAÇÃO JAVA I

- Prof: José Antonio Gonçalves
- zag655@gmail.com

- Ao me enviar um e-Mail coloque o "**Assunto**" começando: "pós2013_2+seu nome"

Nestes Slides:

- **Orientação a Objetos em Java:**

Conceitos sobre Coleções de Objetos;

Contextualizando (1 de 4)

- Até o presente momento foi visto como construir **CLASSES**, que estas são nada mais que tipos de dados definidos pelo usuário (programador) e por isso pode-se chamá-las de **TAD** (Tipo Abstrato de Dados);

- Também foi explicado que um OBJETO é a instância de uma classe. Isto é, a partir do momento em que se declara e instancia uma variável de seu tipo na Memória Principal (RAM). A esta instância chamamos de objeto.

Mas e se precisar trabalhar com um conjunto de objetos?

Contextualizando (2 de 4)

Mas e se precisar trabalhar com um conjunto de objetos?

- A primeira resposta que pode nos vir à mente são os arrays (matrizes e vetores). Pode-se declarar arrays de objetos. Esta seria uma solução, porém esta solução pode levar a outras perguntas:

E se não souber a quantidade de objetos que se deseja manipular? Como definir o tamanho deste array?

Neste ponto podemos ter saudades das tecnologias que permitem a alocação dinâmica da RAM, como os “ponteiros” fornecidos pela linguagem C (por exemplo). Mas... Java não permite este tipo de acesso direto à RAM. Trata-se de uma característica desta tecnologia que, se por um lado garante a segurança do hardware em quanto executa uma aplicação, por outro parece nos tirar a possibilidade de trabalhar com quantidades indefinidas de dados.

Como resolver este tipo de problema?

Contextualizando (3de 4)

E se não souber a quantidade de objetos que se deseja manipular? Como definir o tamanho deste array?

Neste ponto podemos ter saudades das tecnologias que permitem a alocação dinâmica da RAM, como os “ponteiros” fornecidos pela linguagem C (por exemplo). Mas... Java não permite este tipo de acesso, direto à RAM. Esta “proibição” trata-se de uma característica desta tecnologia que, se por um lado garante a segurança do hardware em quanto executa uma aplicação, por outro parece nos tirar a possibilidade de trabalhar com quantidades indefinidas de dados e alocações dinâmicas.

Como resolver este tipo de problema?

Contextualizando (4de 4)

Como resolver este tipo de problema?

A tecnologia Java nos oferece várias soluções para isto por meio de estruturas capazes de armazenar e gerenciar conjunto de objetos. Genericamente este tipo de estrutura chama-se:

Collections

Collections *(definição)*

– Um tipo de **OBJETO** que agrupa um conjunto de elementos (uma estrutura de dados) para facilitar nosso trabalho;

– São **usadas** para armazenar, recuperar e manipular dados.

– Tipicamente representam dados que possuem alguma relação em comum:

- Manipular um conjunto de alunos, um conjunto de jogadores...

– Em suma, é uma forma que temos de trabalhar com qualquer conjunto de dados.

– Possui um conjunto de implementações (**Interfaces e Classes**) oferecidas pelo pacote java.util;

Collections *(definição)*

Quais os benefícios em usar?

– Reduz esforço de programação;

- Algoritmos úteis já estão implementados (ordenação, pesquisa, inclusão...)
- Permite que concentremos esforços em itens importantes de nosso sistema;

– Aumenta a velocidade e qualidade da programação;

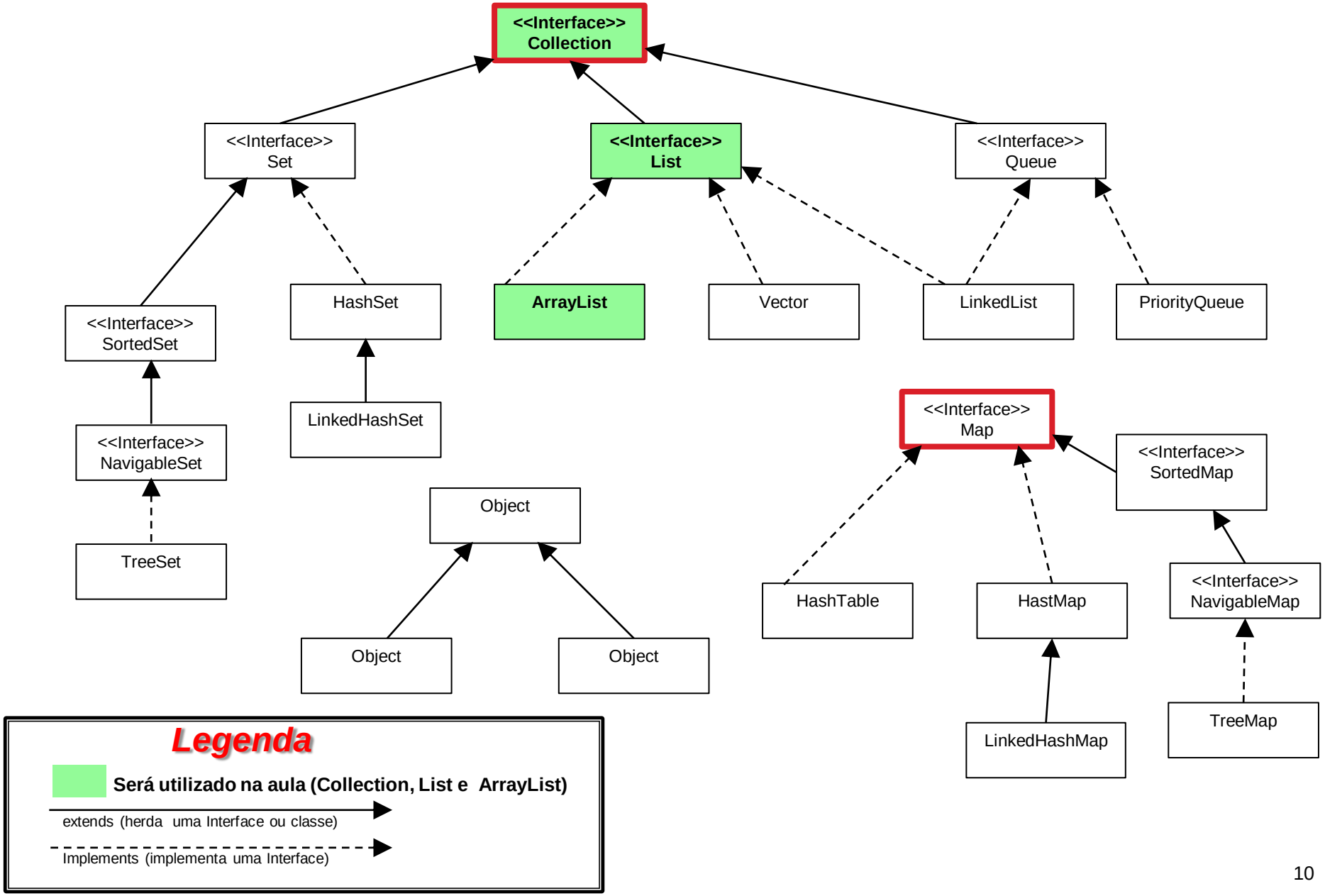
- As implementações já estão bem testadas garantindo qualidade e robustez;
- Existem implementações mais ágeis que outras, veremos a frente;

– Reduz esforços para aprender/conceber novas API´s;

- A forma como estão estruturadas, seguindo a implementação padrão da interface Collection, permite maior facilidade de aprendizado e facilidade na criação de novas implementações.

– Promove reuso de software

Collections *(hierarquias e dependências)*



Collections (tipos)

Set: Está diretamente relacionada com a ideia de conjuntos. Assim como um conjunto, as classes que implementam esta interface não podem conter elementos repetidos.

– **HashSet**: Implementação de Set que utiliza uma tabela hash (a implementação da Sun utiliza a classe HashMap internamente) para guardar seus elementos. Não garante a ordem de iteração, 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 quanto para modificação.

– **LinkedHashSet**: Implementação de Set que estende HashSet, 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 leve queda na performance em relação à HashSet, mas ainda é mais rápida que uma TreeSet, que utiliza comparações para determinar a ordem dos elementos.

Collections (tipos)

Set:

– **SortedSet**: Interface que estende Set, adicionando a semântica de ordenação natural dos elementos. A posição dos elementos ao percorrer a coleção é determinado pelo retorno do método compareTo(o), caso os elementos implementem a interface Comparable, ou do método compare(o1, o2) de um objeto auxiliar que implemente a interface Comparator.

– **TreeSet:** Implementação de SortedSet que utiliza internamente uma TreeMap, que por sua vez utiliza o algoritmo Red-Black para a ordenação da árvore de elementos. Isto garante a ordenação ascendente da coleção, de acordo com a ordem natural dos elementos, definida pela implementação da interface Comparable ou Comparator. Use esta classe quando precisar de um conjunto (de elementos únicos) que deve estar sempre ordenado, mesmo sofrendo modificações. Para casos onde a escrita é feita de uma só vez, antes da leitura dos elementos, talvez seja mais vantajoso fazer a ordenação em uma List, seguida de uma cópia para uma LinkedHashSet (dependendo do tamanho da coleção e do número de repetições de elementos).

Collections (tipos)

List: Interface que extende Collection, e que define coleções ordenadas (sequências), onde se tem o controle total sobre a posição de cada elemento, identificado por um índice numérico. Na maioria dos casos, pode ser encarado como um "array de tamanho variável" pois, como os arrays primitivos, é acessível por índices, mas além disso possui métodos de inserção e remoção;

– **ArrayList:** 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. Em uma inserção no meio da lista (índice < tamanho), o conteúdo posterior ao índice é deslocado em uma posição. 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);

- **LinkedList**: 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 um fila (FIFO - First-In-First-Out), onde os elementos são retirados da lista na mesma sequência em que são adicionados;

Collections (tipos)

List:

-**Vector:** 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 um ambiente multitarefa (acessado por várias threads) sem perigo de perda da consistência de sua estrutura interna.

Em sistemas reais, essa sincronização acaba adicionando um overhead desnecessário, pois mesmo quando há acesso multitarefa à lista (o que não acontece na maioria das vezes), quase sempre é preciso fazer uma nova sincronização nos métodos de negócio, para \'atomizarem\' operações seguidas sobre o Vector.

Collections (tipos)

Interface Map

- A interface Map, auxilia no trabalho com estruturas chave-valor.

- As chaves de um map devem ser únicas.

- Temos algumas implementações de uso geral
  - HashMap
  - TreeMap
  - LinkedHashMap

Collections (tipos)

Interface Map : HashMap

– Esta implementação oferece todas as operações de mapa comuns. Esta classe não faz garantias quanto à ordem do mapa, em particular, não garante que a ordem permanecerá constante ao longo do tempo.

– HashMap não possui ordenação, então os valores ficam todos sortidos na hora da iteração.

– HashMap aceita como chave um valor nulo.

– LinkedHashMap

Collections (tipos)

Interface Map : TreeMap

– A TreeMap utiliza o conceito de Árvore Rubro Negra. Pode se utilizar a implementação da interface comparable no nomento da instanciação de um TreeMap para controlar a ordenação. Esta implementação também oferece os métodos básicos de inserção e remoção no mapa .

– TreeMap ordena os registros à medida em que são inseridos, ou seja, se iterarmos esse mapa já receberemos todos os valores ordenados pela chave.

– TreeMap não aceita como chave um valor nulo.

Collections (tipos)

Classes Utilitárias

– Estas classes oferecem métodos estáticos para auxiliar o trabalho com coleções e arrays(arranjos)

– **Arrays**

• Contém métodos estáticos para classificar, pesquisar, comparar, copiar, redimensionar, converter para String, e preencher arrays de primitivos e objetos.

• Os métodos dessa classe lançam uma NullPointerException, se a referência matriz especificada for nula.

• A documentação para os métodos contidos nesta classe inclui descrição das implementações. Tais descrições devem ser consideradas como notas de execução, em vez de partes da especificação. Os programadores devem se sentir livres para substituir outros algoritmos, desde que a especificação em si seja respeitada.

Collections (tipos)

Classes Utilitárias

– Estas classes oferecem métodos estáticos para auxiliar o trabalho com coleções e arrays(arranjos)

– **Collections**

• Esta classe é composta exclusivamente de métodos estáticos que operam ou retornam coleções. Ela contém algoritmos polimórficos que operam em coleções,e ou retornam uma nova coleção.

• Os métodos desta classe lançam uma NullPointerException se as coleções ou objetos de classe que lhes são prestados forem nulos.

Collections (tipos)

Interfaces Auxiliares

– Iterator

• É uma boa alternativa para iteração com estrutura de dados;

• Define as operações básicas para o percorrimento dos elementos da coleção.

• Utiliza o pattern de mesmo nome (Iterator, GoF), desacoplando o código que utiliza as coleções de suas estruturas internas.

• É possível remover elementos da coleção original utilizando o método remove(), que remove o elemento atual da iteração, mas esta operação é de implementação opcional, e uma UnsupportedOperationException será lançada caso esta não esteja disponível:

```
Map mapa = new HashMap();
terator itChaves = mapa.keySet().iterator();
while (itChaves.hasNext()) {
Object chave = itChaves.next();
Object valor = mapa.get(chave);
System.out.println(chave + " = " + valor);
}
```

Collections (tipos)

Interfaces Auxiliares

- ListIterator ;
- Comparable ;
- Comparator ;
- Enumeration ;
- RandonAcces .

Bibiografia

Anotações da apresentação dos alunos de Programa de Pós Graduação em Informática (PPGI) da UTFPR – Campus Cornélio Procópio-PR:

Disciplina: Programação Orientada a Objetos

Professor: Dr. Fernando Barreto

Alunos mestrandos:

Devair Aparecido Canedo da Silva Junior;

Ricardo F. P. Satin;

Wellington Carvalho.