Coleções genéricas

Prof. Ítalo Assis

Ajude a melhorar este material =]

Encontrou um erro? Tem uma sugestão?

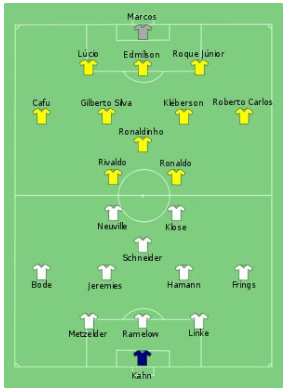
Envie e-mail para italo.assis@ufersa.edu.br

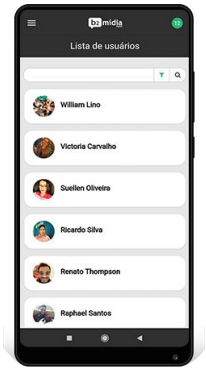
Agenda

- Coleções
- Classes empacotadoras de tipo
  - *Autoboxing* e *auto-unboxing*
- *Interface Collection* e classe *Collections*
- Listas
- Pilhas
- Filas
- Conjuntos
- Mapas

Visão geral

- Vimos anteriormente a coleção *ArrayList* genérica
  - uma estrutura de dados dinamicamente redimensionável do tipo *array*, que armazena referências a objetos de um tipo que você especifica ao criar o *ArrayList*
- Agora, discutiremos o *framework collection* do Java, que contém muitas outras estruturas de dados genéricos predefinidas

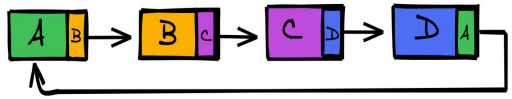


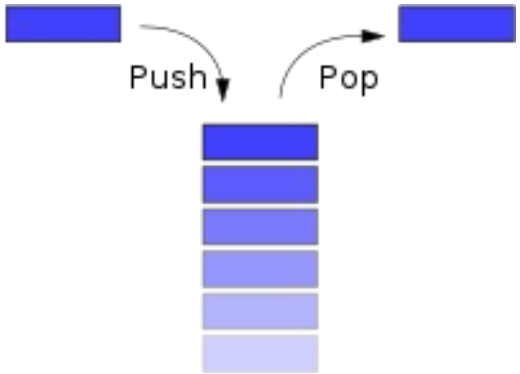


Coleções

- Uma coleção é um objeto que pode armazenar referências a outros objetos
- As classes e interfaces da estrutura das coleções são membros do pacote *java.util*

Collection	A interface-raiz na hierarquia de coleções a partir da qual as interfaces *Set*, *Queue* e *List* são derivadas
Set	Uma coleção que não contém duplicatas
List	Uma coleção ordenada que pode conter elementos duplicados
Map	Uma coleção que associa chaves a valores e que não pode conter chaves duplicadas. *Map* não deriva de *Collection*
Queue	Em geral, uma coleção “primeiro a entrar, primeiro a sair” que modela uma fila de espera; outras ordens podem ser especificadas
...	...





(Relembrando) classes empacotadoras de tipo

- Coleções genéricas não podem manipular variáveis de tipos primitivos
  - Os tipos primitivos não têm métodos
- Os tipos primitivos tem uma classe empacotadora de tipo correspondente
  - Essas classes chamam-se *Boolean*, *Byte*, *Character*, *Double*, *Float*, *Integer*, *Long* e *Short*
  - Classes empacotadoras permitem manipular valores de tipo primitivo como objetos
  - Os métodos relacionados a um tipo primitivo estão localizados na classe empacotadora de tipo correspondente
    - Exemplo: o método *parseInt*, que converte uma *String* em um valor *int*, está localizado na classe *Integer*

Autoboxing e auto-unboxing

- Conversões automáticas entre valores de tipo primitivo e objetos empacotadores de tipo
  - *Autoboxing*: tipo primitivo objeto em pacotador de tipo
  - *Auto-unboxing*: objeto em pacotador de tipo tipo primitivo

```
Integer[] integerArray = new Integer[5];

integerArray[0] = 10;

int value = integerArray[0];
```

Interface *Collection* e classe *Collections*

- Interface *Collection*
  - Contém operações realizadas na coleção inteira
    - adicionar, limpar, comparar objetos, determinar tamanho, determinar se a coleção está vazia...
  - Poder ser convertida em um *array*
  - Fornece um método que retorna um objeto *Iterator*
    - permite percorrer a coleção e remover seus elementos durante a iteração
- A classe *Collections* fornece métodos *static* que pesquisam, classificam e realizam outras operações sobre as coleções

```
// Exemplo de iterator
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        ArrayList<Double> notas = new ArrayList<>(Arrays.asList(5.5, 6.5, 7.5, 8.5));
        Iterator<Double> iterator = notas.iterator();

        imprimeCollection(notas);
        while (iterator.hasNext()) {
            if (iterator.next() < 7.0) {
                iterator.remove();
            }
        }
        imprimeCollection(notas);
    }

    private static void imprimeCollection(Collection<Double> collec) {
        for (Double elemento : collec) {
            System.out.print(elemento + " ");
        }
        System.out.println();
    }
}
```

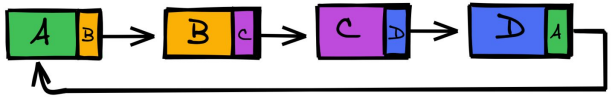
Métodos de coleções

sort	Classifica os elementos de uma *List*
binarySearch	Localiza um objeto em uma *List* usando o algoritmo de pesquisa binária
reverse	Inverte os elementos de uma *List*
shuffle	Ordena aleatoriamente os elementos de uma *List*
fill	Configura todo elemento *List* para referir-se a um objeto especificado
copy	Copia referências de uma *List* em outra
min	Retorna o menor elemento em uma *Collection*
max	Retorna o maior elemento em uma *Collection*
addAll	Acrescenta todos os elementos em um *array* a uma *Collection*
frequency	Calcula quantos elementos da coleção são iguais ao elemento especificado
disjoint	Determina se duas coleções não têm nenhum elemento em comum

Prática

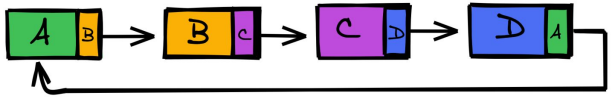
- Crie um método que recebe uma *ArrayList* de *Strings* ordenada e uma *String* alvo e informa se a *String* alvo está na *ArrayList* e qual a posição da sua primeira ocorrência
- No método principal:
  - Crie uma *ArrayList* contendo nomes de cores não ordenados alfabeticamente e a imprima
  - Ordene e imprima novamente a *ArrayList*
  - Teste o método criado anteriormente para diferentes cores

Listas



- Uma *List* é uma *Collection* ordenada (cada elemento tem um índice) que pode conter elementos duplicados
  - o índice do primeiro elemento é zero
- Fornece métodos para:
  - manipular elementos por meio de seus índices
  - manipular um intervalo especificado de elementos
  - procurar elementos
  - obter um *ListIterator* para acessar os elementos
- A interface *List* é implementada por várias classes, inclusive as classes *ArrayList*, *LinkedList* e *Vector*

Listas



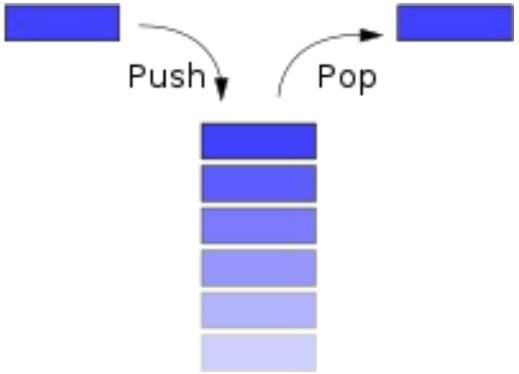
- *ArrayList* e *Vector* são implementações de *arrays* redimensionáveis
  - Inserir um elemento entre os elementos existentes de uma *ArrayList* ou *Vector* é uma operação ineficiente
  - *Vector*:
    - Sincronizado;
    - Classe legada;
    - Incrementa tamanho 100%
- Uma *LinkedList*:
  - Permite a inserção (ou remoção) eficiente dos elementos no meio de uma coleção
  - É menos eficiente que uma *ArrayList* para pular para um elemento específico na coleção

Prática

- Crie um método que recebe duas *LinkedLists* de *Strings* e, utilizando um *Iterator*, remove da primeira os elementos que aparecem na segunda
- No método principal:
  - crie duas *LinkedLists* de *Strings*, cada uma contendo uma lista de cores
  - utilize o método criado para remover da primeira lista as cores que aparecem na segunda lista
  - imprima a primeira lista antes e depois das exclusões

Pilhas

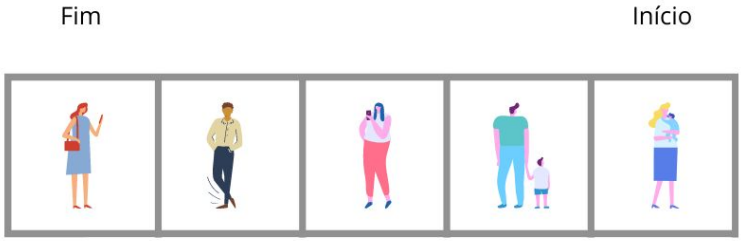
- A classe *Stack* estende a classe *Vector* para implementar uma estrutura de dados de pilha
- Métodos da classe *Stack*:
  - *push*: adiciona um elemento no topo da pilha
  - *pop*: remove um elemento no topo da pilha
  - *peek*: retorna o elemento no topo da pilha sem remover o elemento da pilha
  - *isEmpty*: herdado da classe *Vector*, determina se a pilha está vazia



Prática

- Crie um programa que contém uma pilha que armazena elementos de qualquer tipo numérico
- Adicione elementos de tipos numéricos variados
- Remova os elementos um a um até esvaziar a pilha
- Após cada alteração, imprima a pilha

Filas

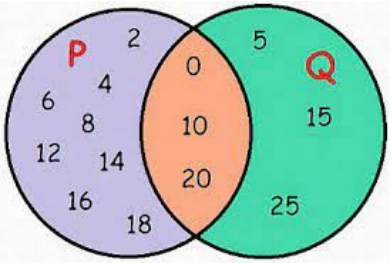


- A interface *Queue* estende a interface *Collection* e fornece operações adicionais para inserção, remoção e inspeção de elementos em uma fila
- *PriorityQueue*, que implementa a interface *Queue*, ordena elementos por sua ordem natural
    - Ao adicionar elementos a uma *PriorityQueue*, os elementos são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade (isto é, o menor valor) será o primeiro elemento removido da *PriorityQueue*.
- Algumas operações: *offer* (inserir), *poll* (remover), *peek*, *clear* e *size*.

Prática

- Crie um programa que contém uma fila de prioridades de *Doubles*
- Adicione elementos a esta fila
- Remova e apresente os elementos um a um até zerar a fila

Conjuntos



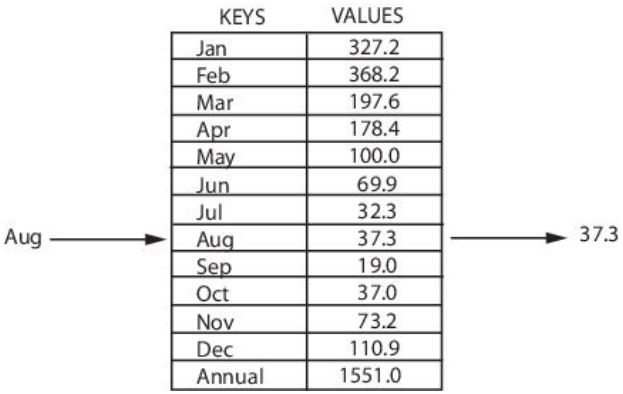
- Um *Set* é uma *Collection* não ordenada de elementos únicos
- A estrutura de coleções contém diversas implementações de *Set*
  - *HashSet* armazena seus elementos em uma tabela de *hash*
  - *TreeSet* armazena seus elementos em uma árvore
- *SortedSet* é uma interface e mantém seus elementos ordenados
    - A classe *TreeSet* implementa *SortedSet*
    - Alguns métodos:
      - *headSet* (elementos menores), *tailSet* (elementos maiores), *first* e *last*

Prática

- Escreva um programa que crie e inicialize um conjunto de cores utilizando um *TreeSet*
  - Experimente inserir a mesma cor mais de uma vez
- Imprima o conjunto criado
- Escolha uma cor e, considerando a ordem alfabética, apresente:
  - uma lista com as cores que vem antes da cor selecionada
  - uma lista das cores a partir da cor selecionada
  - o primeiro e o último elementos do conjunto

Mapas

- Associam chaves a valores
- As chaves em um *Map* devem ser únicas, mas os valores associados não precisam ser
- Exemplos de classes que implementam a interface Map: *Hashtable*, *HashMap*, *SortedMap (interface)* e *TreeMap*
  - *Hashtable* e *HashMap* armazenam elementos em tabelas de *hash*
  - *TreeMaps* armazenam elementos em árvores
  - *TreeMap* é uma implementação de *SortedMap* (mantém as chaves ordenadas)



Prática

- Escreva um programa que utiliza um *HashMap* para contar e exibir o número de ocorrências de cada palavra em uma *String*
  - Considere como iguais palavras que se diferenciam apenas por letras maiúsculas/minúsculas
  - Exibir palavras em ordem alfabética

Os códigos relacionados a esta aula estão disponíveis em

https://github.com/italoaug/Programacao-Orientada-a-Objetos/blob/main/codigos/colecoes

Referências

SANTOS, R. **Introdução à programação orientada a objetos usando JAVA**. 2. ed. Rio de Janeiro: Campus, 2013. 336p.

DEITEL, Paul; DEITEL, Harvey. **Java: como programar**. 10. ed. São Paulo: Pearson Education do Brasil, 2017.