Java – Aula 17
Collections

Cristiano Amaral Maffort

cristiano@cefetmg.br

maffort@gmail.com

Técnico em Informática

Departamento de Computação

CEFET-MG – Belo Horizonte

Interface de Programação de Aplicações

- Uma API (*Application programming interface*) é, grosso modo, um ***conjunto de métodos***/funções
  - estabelecidos e ***disponibilizados*** por um componente de software
  - ***para utilização*** de suas funcionalidades
  - por usuários/aplicativos cujo ***maior interesse é pelos serviços*** oferecidos pelo componente,
    - e não nos detalhes de implementação

Interface de Programação de Aplicações

- O propósito das APIs é o de ***simplificar o desenvolvimento*** de programas e aplicações
  - Fazendo com que ***o programador conheça apenas a interface*** das funcionalidades da API que está incorporando ao programa
    - Para isso, por óbvio, é necessário que o desenvolvedor conheça os objetivos, responsabilidades e comportamentos dessas funcionalidades.

Interface de Programação de Aplicações

- As APIs ***abstraem*** elementos que compõem o software
  - Permitindo que o desenvolvedor não precise conhecer detalhes de implementação de todas as partes que compõem o sistema
    - Novamente → precisará saber como utilizar as funcionalidades/serviços fornecidos pela API
      - E como se dará a interação destas com os outros elementos de seu software

Collections

- A API de coleções de Java possui um conjunto de classes que oferecem ***formas diferentes de colecionar dados*** e que levam em consideração
  - Eficiência e forma no acesso, inserção, remoção e pesquisa
  - Organização dos dados
- São exemplos de estruturas de dados
  - Vetores/Arrays
  - Listas (incluindo Fila, Pilha, Deque)
  - Conjuntos
  - Mapas / Tabelas Hash
  - Árvores

Java Collections API

- Apesar de Java oferecer uma robusta API de coleções
  - Certas operações poderão ter um desempenho melhor ou pior
    - Dependendo do tipo da coleção utilizada
  - Certas operações poderão ter restrições
    - ou mesmo funcionalidade especial
    - Dependendo do tipo da coleção utilizada

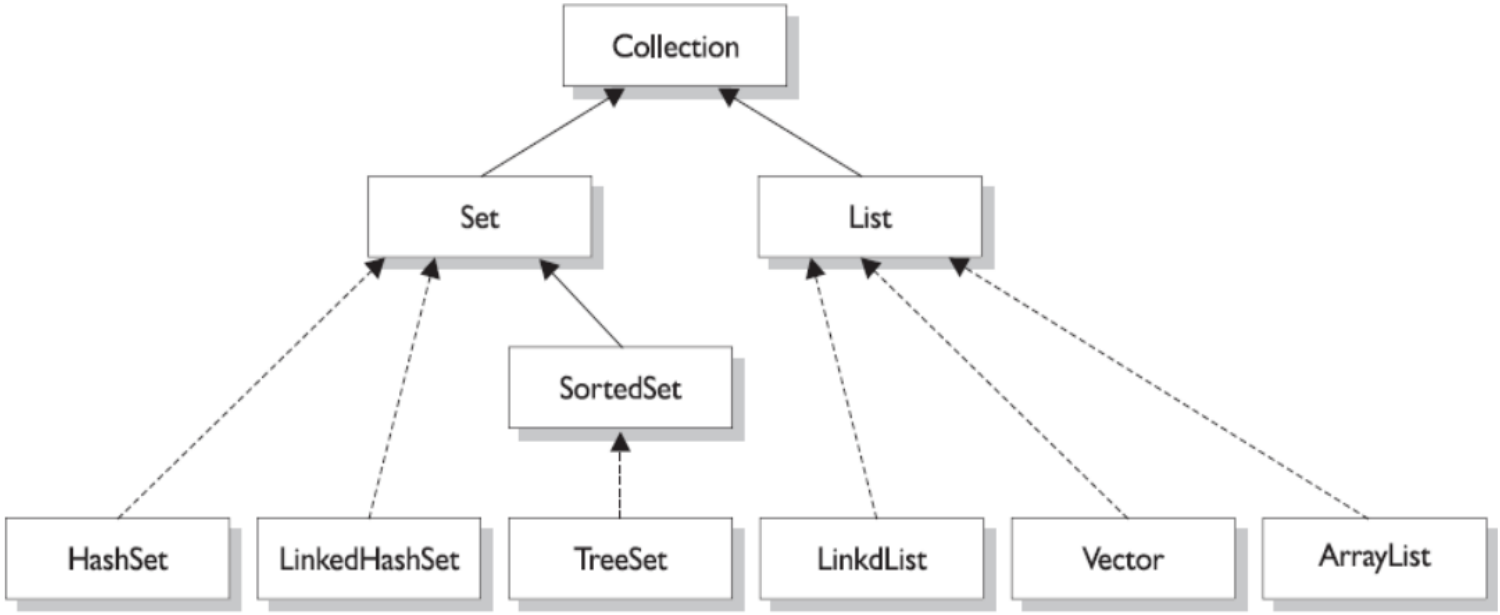
Java Collections API

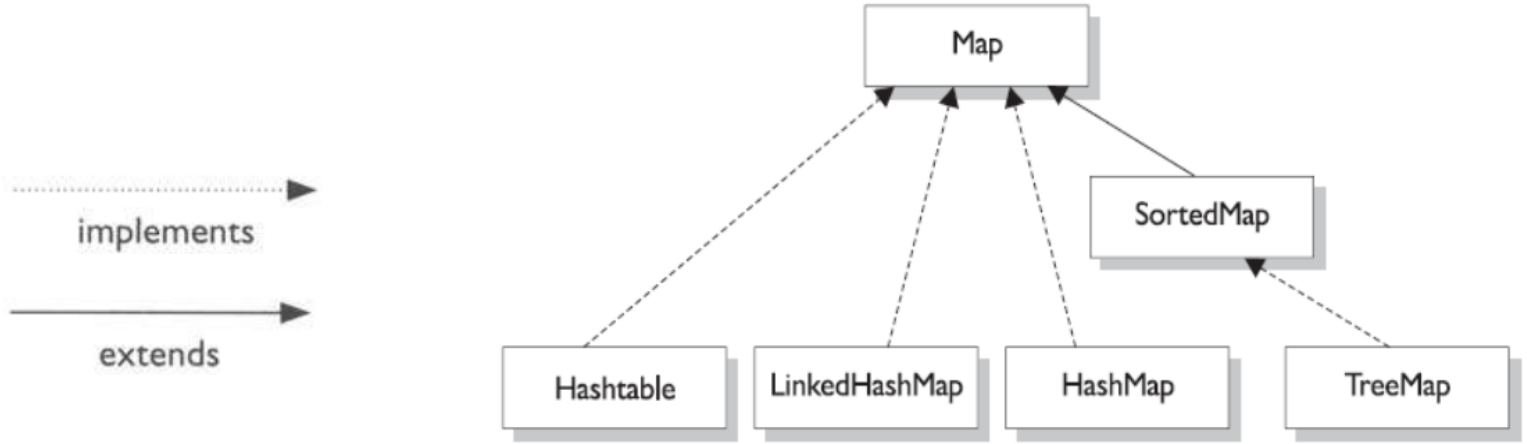
- Na prática, a API de coleções de Java
  - Estão localizadas no pacote java.util
  - Implementa as principais estruturas de dados
    - Usando apenas duas interfaces
  - Oferece implementações de iteradores
    - Implementações do padrão Iterator que permitem que o usuário das estruturas possam realizar algumas operações genéricas sobre as estruturas, sem precisar conhecer detalhes de funcionamento da estrutura.
  - Oferece classes utilitárias
    - Acessórias às estruturas de dados
    - Que oferecem funcionalidades para manipulação das coleções

Tipos de Coleções

- Basicamente, as coleções são classificadas em três grupos
  - Listas (List)
  - Conjuntos (Set)
  - Mapas (Map)

Tipos de Coleções



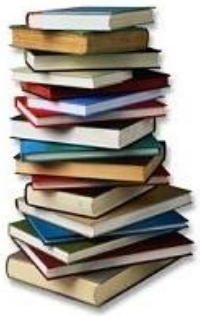


Coleções mais comuns

- **Fila (Queue)**
  - FIFO → primeiro elemento a entrar é o primeiro elemento a sair
    - As inserções são feitas no final
    - As remoções são feitas no início
  - Consultas são feitas desenfileirando os elementos, um a um, até encontrar o elemento desejado
    - Ou até esvaziar a fila sem que o elemento seja encontrado
- Aplicações:
  - Quando é necessário organizar acessos a recursos limitados
    - Ex: fila de impressão, alocação de recursos a processos, roteamento de pacotes, gravação de dados em mídia, etc

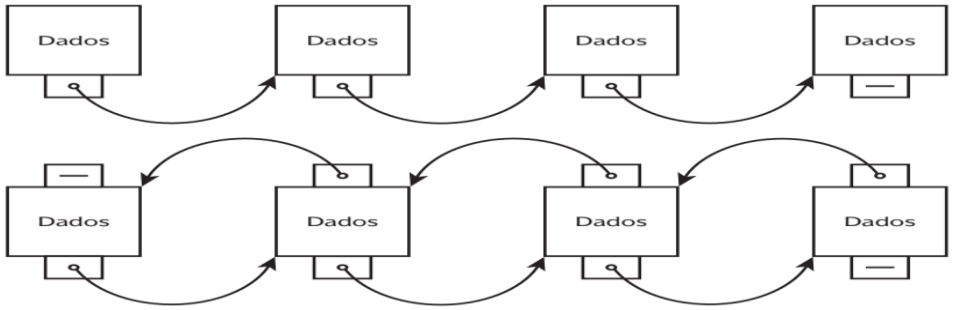
Coleções mais comuns

- **Pilha (Stack)**
  - LIFO → último elemento a entrar é o primeiro elemento a sair
    - As inserções e remoções são feitas no topo da pilha
  - Consultas são feitas desempilhando os elementos, um a um, até encontrar o elemento desejado
    - Ou até esvaziar a pilha sem que o elemento seja encontrado
- Aplicações:
  - Mais comum em programação de algoritmos
    - Ex: avaliação de expressões, parsing sintático, fluxos de execução, recursividade, backtracking, busca em largura e profundidade, etc.



Coleções mais comuns

- **Lista (List)**
  - Pilhas e filas possuem regras rigorosas que definem a ordem em que os dados podem ser acessados.
  - As listas são mais flexíveis
    - Seus elementos podem ser acessados de diversas maneiras.
- As listas podem ser implementadas de diversas formas
  - Por meio de Array (ArrayList e Vector)
  - Simplesmente ou duplamente encadeadas (LinkedList)



Coleções mais comuns

- **Mapas (Map)**
  - Utiliza uma tupla chave-valor que utiliza, na estrutura, a chave para produzir um índice de acesso ao valor
  - As chaves são geralmente produzidas a partir do valor, por meio de um algoritmo de hash (espalhamento).
- **HashMap**
  - Não garante ordem do mapeamento e nem que continuará constante ao longo do tempo
- **TreeMap**
  - Adiciona semântica de ordenação dos elementos
  - Utiliza a árvore Red-Black (Rubro-Negra) para ordenação dos elementos

Coleções mais comuns

- **Conjuntos (Set)**
  - Como em um conjunto, não admitem repetição dos seus elementos.
- **HashSet**
  - Utiliza uma tabela hash para armazenar seus elementos
  - Acesso ao dado é extremamente rápido.
  - Não garante a ordem de iteração, nem que a ordem será mantida no tempo.
- **LinkedHashSet**
  - Semelhante a HashSet, mas adiciona previsibilidade à ordem de iteração sobre os elementos (mantém ordem de inserção)
  - Implementada usando uma lista duplamente encadeada.

Coleções mais comuns

- **Conjuntos (Set)**
  - Como em um conjunto, não admitem repetição dos seus elementos.
- **TreeSet**
  - Implementa SortedSet → adiciona semântica de ordenação dos elementos
  - Utiliza a árvore Red-Black para ordenação dos elementos
  - Os elementos estarão sempre ordenados, mesmo sofrendo modificações.

Classes Utilitárias

- Oferecem métodos estáticos para auxiliar o trabalho com coleções

- **Arrays**
  - Contém métodos para copiar, redimensionar, classificar, pesquisar, comparar (entre outros) elementos de um array.

- **Collections**
  - Funções semânticas similares às da classe Arrays
    - Entretanto, seus métodos que operam sobre coleções, apenas.

Classes Auxiliares

- **Iterator**
  - Define operação para percorrimento dos elementos de uma coleção
    - Desacoplando o algoritmo de percorrimento da estrutura interna
  - Em algumas estruturas, durante o percorrimento, pode-se realizar operações de remoção ou alteração
    - Se o operação não for suportada, `UnsupportedOperationException` será lançada.

Classes Auxiliares

- **Iterator - Exemplo**

```
public static void main(String[] args) {

  Map<String, String> capitalCities = new HashMap<String, String>();

  // país e cidade são chave e valor, respectivamente
  capitalCities.put("Brasil", "Brasília");
  capitalCities.put("England", "London");
  capitalCities.put("Germany", "Berlin");
  capitalCities.put("Norway", "Oslo");
  capitalCities.put("USA", "Washington DC");

  Iterator it = capitalCities.keySet().iterator();

  while (it.hasNext()) {
      String key = (String) it.next();
      String value = capitalCities.get(key);
      System.out.println(key + " = " + value);
  }
}
```

Classes Auxiliares

- **Iterator**
  - Define operação para percorrimento dos elementos de uma coleção
    - Desacoplando o algoritmo de percorrimento da estrutura interna
  - Em algumas estruturas, durante o percorrimento, pode-se realizar operações de remoção ou alteração
    - Se o operação não for suportada, `UnsupportedOperationException` será lançada.

Classes Auxiliares

- **Comparable<T>**
  - Determina ordem de classificação entre dois objetos
  - É uma interface que possui um único método:
    - `int compareTo(T o);`
  - Compara **objetos que tem ordem natural**

```
public class ComparableEx {
    public static void main(String[] args) {
        String str1 = "Letícia";
        String str2 = "Victor";

        if (str1.compareTo(str2) < 0)
            System.out.println(str1 + " vem primeiro.");
        else
            System.out.println(str2 + " vem primeiro.");
    }
}
```

Classes Auxiliares

- **Comparable<T>**

```java
@Override
public int compareTo(Pessoa o) {
    if (this.cpf < o.cpf)
        return -1;
    else if (this.cpf > o.cpf)
        return 1;

    return 0;
}

public static void main(String[] args) {
    List<Pessoa> pessoas = new ArrayList<>();

    pessoas.add(new Pessoa(23456L));
    pessoas.add(new Pessoa(56L));
    pessoas.add(new Pessoa(3456L));
    pessoas.add(new Pessoa(123456L));
    pessoas.add(new Pessoa(456L));

    Collections.sort(pessoas);

    for(Pessoa p: pessoas){
        System.out.println(p.getCPF());
    }
}
```

Classes Auxiliares

- **Comparator<T>**

```java
public class PessoaComparator implements Comparator<Pessoa> {

    @Override
    public int compare(Pessoa p1, Pessoa p2) {
        return p1.compareTo(p2);
    }

    public static void main(String[] args) {
        List<Pessoa> pessoas = new ArrayList<>();

        pessoas.add(new Pessoa(23456L));
        pessoas.add(new Pessoa(56L));
        pessoas.add(new Pessoa(3456L));
        pessoas.add(new Pessoa(123456L));
        pessoas.add(new Pessoa(456L));

        Collections.sort(pessoas, new PessoaComparator());

        for (Pessoa p : pessoas) {
            System.out.println(p.getCPF());
        }
    }
}
```

Classes Auxiliares

- **Object.equals(Object obj)**
  - Avalia a igualdade entre dois objetos, o `this` e o `obj`

```java
public class Pessoa {
    private Long cpf;

    // ...

    @Override
    public boolean equals(Object p) {
        if (cpf == null)
            return false;

        return ((p instanceof Pessoa) &&
                this.cpf.equals(((Pessoa)p).cpf));
    }
}
```

Classes Auxiliares

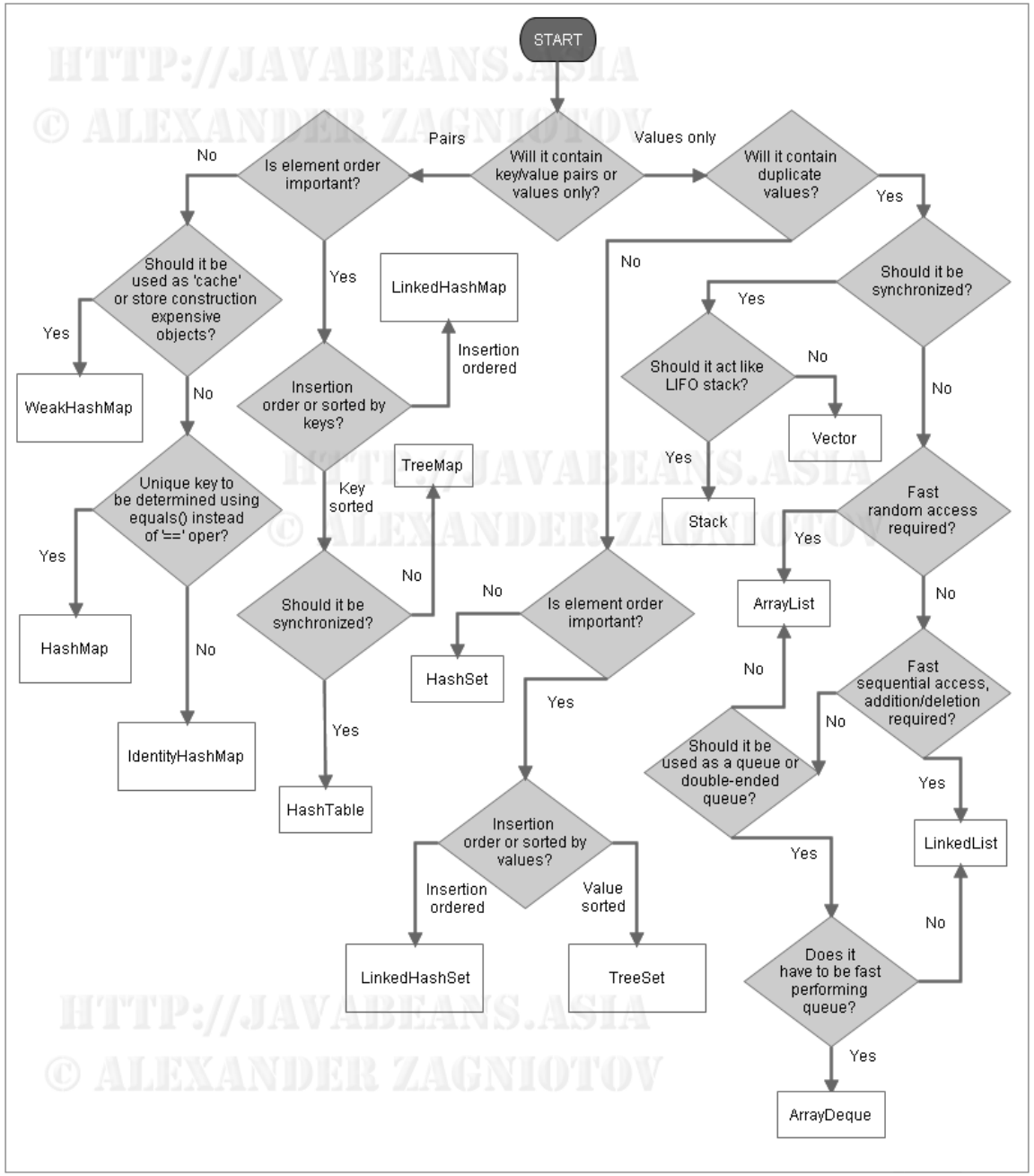
- **Object.hashCode()**
  - Produz uma chave a partir do objeto (this)
  - Deve ser igual para objetos iguais
  - Está ligado ao teste de igualdade (equals)
    - Ao sobrescrever `equals`, deverá também sobrescrever `hashCode`

```java
public class Pessoa {
    private Long cpf;

    // ...

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 29 * hash + Objects.hashCode(this.cpf);
        return hash;
    }
}
```

Qual coleção usar?



Fonte: https://fernandofranzini.files.wordpress.com/2012/06/collections.png

Vocês que lutem!

Bibliografia Obrigatória

- HORSTMANN, Cay S.; CORNELL, Gary. ***Core Java: Fundamentos***. 8. ed. São Paulo: Pearson Prentice Hall, 2020, p. 295 – 326.
- SCHILDT, Herbert; SKRIEN, Dale. ***Programação com Java: uma introdução abrangente***. Porto Alegre: AMGH, 2013, p. 911-964.