Programação Orientada a Objetos

Collections - Java

Coleções – Conceito Geral

Uma coleção é uma estrutura de dados que permite armazenar vários objetos.

Em Java , a coleção também é um objeto.

As operações que podem ser feitas em coleções variam mas normalmente incluem:

- Adição de elementos;
- Remoção de elementos;
- Acesso aos elementos;
- Pesquisa de elementos;

Coleções – Tipos

Dependendo da forma de fazer as 4 operações básicas (adição, remoção, acesso e pesquisa), teremos vários tipos de coleções

Os três grandes tipos de coleções são:

Lista (também chamado de “sequência“);

Conjunto;

Mapa(também chamado de “dicionário“).

Coleções – Tipos

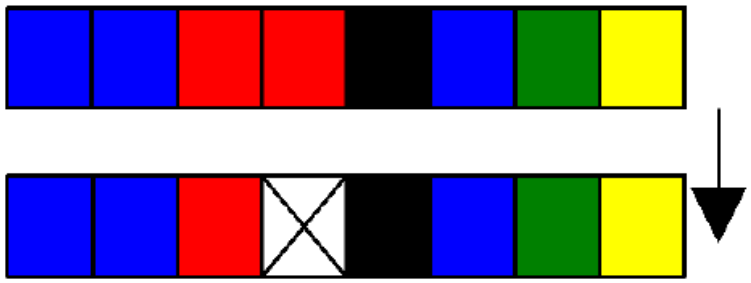
Manipular arrays é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:

Não podemos redimensionar um *array* em Java;

É impossível buscar diretamente por um determinado elemento cujo índice não se sabe;

Não conseguimos saber quantas posições do array já foram “populadas”

Coleções



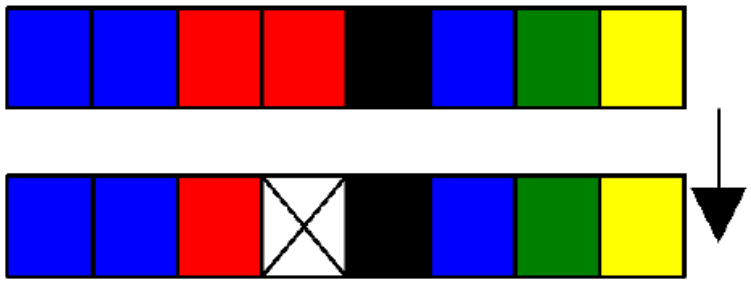
Retire a quarta Conta

conta[3] = **null**;

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco?

- Precisaremos procurar por um espaço vazio?
- Guardaremos em alguma estrutura de dados externa, as posições vazias?
- E se não houver espaço vazio?
- Teríamos de criar um array maior e copiar os dados do antigo para ele?

Coleções



Retire a quarta Conta

conta[3] = **null**;

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco?

- Como posso saber quantas posições estão sendo usadas no array?
- Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

- Com esses e outros objetivos em mente, a Sun criou um conjunto de classes e interfaces conhecido como Collections Framework

Coleções

A **API** do **Collections** é robusta e possui diversas classes que representam estruturas de dados avançadas.

- Por exemplo, não é necessário reinventar a roda e criar uma lista ligada, mas sim utilizar aquela que a Sun disponibilizou.

Listas: Java.Util.List

Um primeiro recurso que a API de Collections traz são **listas.**

Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.

- Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida hora da inserção dos mesmos.

- Lista resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho “infinito”,...).

Esse código já está pronto!

Listas: Java.Util.List

A API de Collections traz a **interface** java.util.List, que específica o que uma classe deve ser capaz de fazer para ser uma lista.

Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

- A implementação mais utilizada da interface List é a **ArrayList**, que trabalha com um array interno para gerar uma lista.
  - **ArrayList** é mais rápida na pesquisa do que sua concorrente.
  - **LinkedList** é mais rápida na inserção e remoção de itens nas pontas.

Listas: Java.Util.List

ArrayList não é um Array!

É comum confundirem uma ArrayList com um array, porém ela não é um array:

O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados.

Porém este atributo está propriamente encapsulado e você não tem como acessá-lo.

Repare, também, que você **não** pode usar [] com uma ArrayList, nem acessar atributo **length**. Não há relação!

Listas: Java.Util.List

Para criar um ArrayList, basta chamar o construtor:

```
ArrayList lista = new ArrayList();
```

É sempre possível abstrair a lista a partir da interface List:

```
List lista = new ArrayList();
```

Para criar uma lista de nomes (String), podemos fazer:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```

Listas: Java.Util.List

A **interface List** possui dois métodos add:

recebe o objeto a ser inserido e o coloca no final da lista

```
lista.add("Manoel");
```

permite adicionar o elemento em qualquer posição da lista

```
lista.add(2,"Manoel");
```

Note que, em momento algum, dizemos qual é o tamanho da lista

Listas: Java.Util.List

Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível.

Todos os métodos trabalham com Object.

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(100);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

List contas = new ArrayList();

contas.add(c1);
contas.add(c2);
```

Listas: Java.Util.List

Para saber quantos elementos há na lista, usamos o método size():

System.out.println(contas.size());

Há ainda um método get(int) que recebe como argumento o índice do elemento que se quer recuperar. Através dele, podemos fazer um for para iterar na lista de contas:

```
for (int i = 0; i < contas.size(); i++) {
   contas.get(i); // código não muito útil....
}
```

Mas como fazer para imprimir o saldo dessas contas?

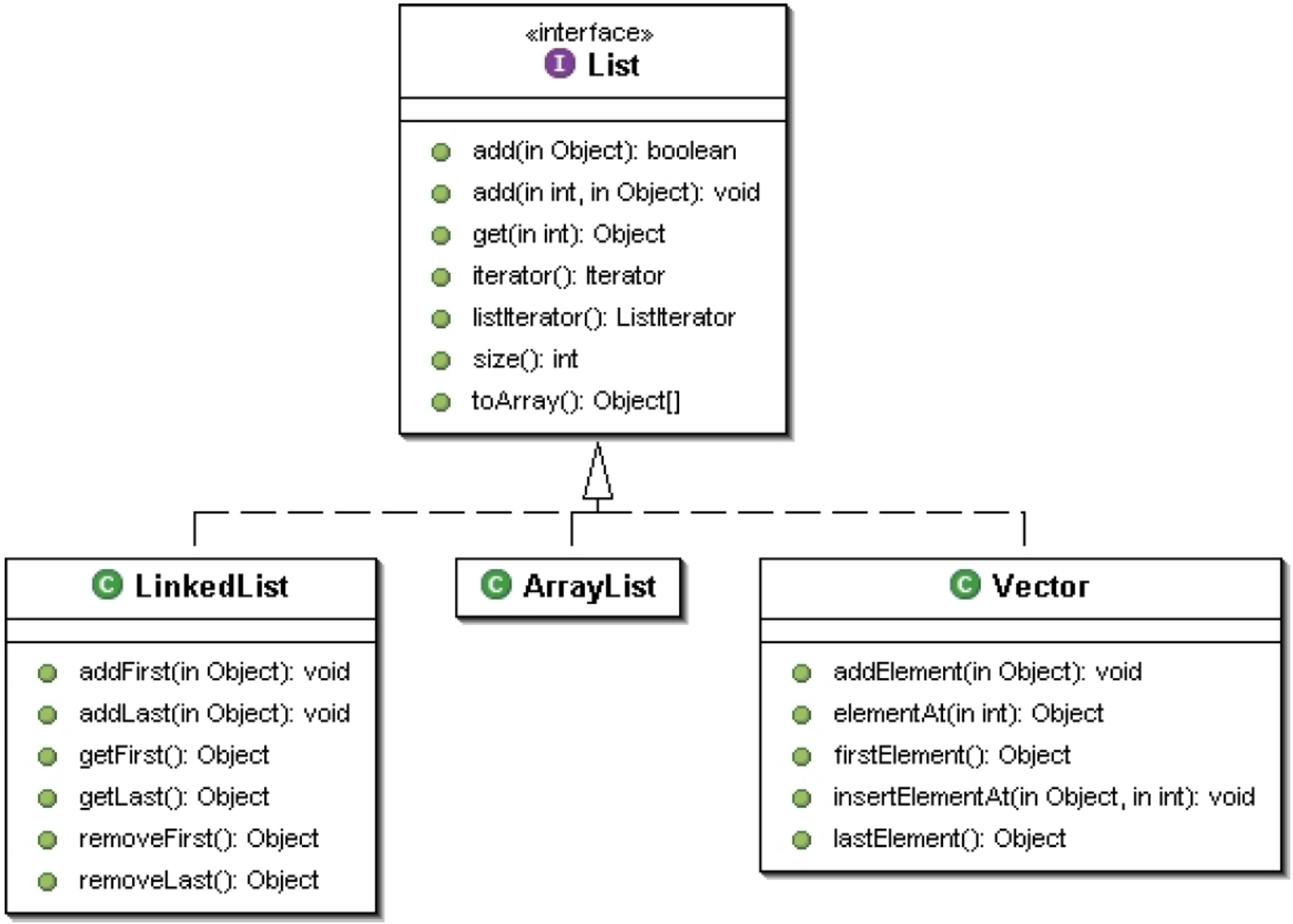
Listas: Java.Util.List

Podemos acessar o getSaldo() diretamente após fazer contas.get(i)?

Não podemos; lembre-se que toda lista trabalha sempre com Object. Assim, a referência devolvida pelo get(i) é do tipo Object, sendo necessário o cast para ContaCorrente se quisermos acessar o getSaldo()

```
for (int i = 0; i < contas.size(); i++) {
 ContaCorrente cc = (ContaCorrente) contas.get(i);
 System.out.println(cc.getSaldo());
}// note que a ordem dos elementos não é alterada
```

Listas: Java.Util.List



Listas: Java.Util.List

Accesso Aleatório e Percorrendo Listas com Get

Algumas listas, como a ArrayList, têm acesso aleatório aos seus elementos:

A busca por um elemento em uma determinada posição é feita de maneira imediata, sem que a lista inteira seja percorrida (que chamamos de acesso sequencial).

Neste caso, o acesso através do método get(int) é muito rápido.

Caso contrário, percorrer uma lista usando um for como esse que acabamos de ver, pode ser desastroso.

Listas: Java.Util.List

Uma lista é uma excelente alternativa a um array comum

- Temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

A outra implementação muito usada, a LinkedList

- Fornece métodos adicionais para obter e remover o primeiro e último elemento da lista.

- Também tem o funcionamento interno diferente, o que pode impactar performance, como veremos em alguns exercícios

Listas no Java 5 e Java 7 com Generics

Em qualquer lista, é possível colocar qualquer Object. Com isso, é possível misturar objetos:

```
ContaCorrente cc = new ContaCorrente();

List lista = new ArrayList();
lista.add("Uma string");
lista.add(cc);
...
```

Mas e depois, na hora de recuperar esses objetos?

Como o método get devolve um Object, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

Listas no Java 5 e Java 7 com Generics

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

Repare no uso de um parâmetro ao lado de List e ArrayList:

Indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo ContaCorrente.

Isso nos traz uma segurança em tempo de compilação:

```
contas.add("uma string"); // isso não compila mais!!
```

O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo ContaCorrente

A importância das Interfaces nas Coleções

```
class Agencia {
    public ArrayList<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<Conta>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

Para que precisamos retornar a referência específica a uma ArrayList? - Para que ser tão específico?

A importância das Interfaces nas Coleções

```
class Agencia {
    public ArrayList<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<Conta>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

Para que precisamos retornar a referência específica a uma ArrayList? - Para que ser tão específico?

Dessa maneira, o dia que optarmos por devolver uma LinkedList em vez de ArrayList, as pessoas que estão usando o método buscaTodasContas poderão ter problemas

A importância das Interfaces nas Coleções

```
class Agencia {

    // modificação apenas no retorno:
    public List<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

É o mesmo caso de preferir referenciar aos objetos com InputStream como fizemos na aula passada

Ordenação: Collections.Sort

Vimos que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.

A classe Collections traz um método estático sort que recebe um List como argumento e o ordena por ordem crescente. Por exemplo:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");
System.out.println(lista);
Collections.sort(lista);
System.out.println(lista);
```

Listas: Java.Util.List

Mas toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, ContaCorrente.

E se quisermos ordenar uma lista de ContaCorrente? Em que ordem a classe Collections ordenará?

Pelo saldo? Pelo nome do correntista?

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(500);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c2);
Collections.sort(contas); // qual seria o critério para esta ordenação?
```

É necessário instruir o sort sobre como comparar nossas ContaCorrente a fim de determinar uma ordem na lista.

Listas: Java.Util.List

Para isto, o método sort necessita que todos seus objetos da lista sejam comparáveis e possuam um método que se compara com outra ContaCorrente

Como é que o método sort terá a garantia de que a sua classe possui esse método?

- Isso será feito, novamente, através de um contrato, de uma interface!

- Vamos fazer com que os elementos da nossa coleção implementem a interface java.lang.Comparable, que define o método int compareTo(Object).

Listas: Java.Util.List

Para ordenar as Contas Correntes por saldo, basta implementar o **Comparable**:

```
public class ContaCorrente extends Conta implements
    Comparable<ContaCorrente> {
    // ... todo o código anterior fica aqui

    public int compareTo(ContaCorrente outra) {
    if (this.saldo < outra.saldo) {
    return -1;
    }
    if (this.saldo > outra.saldo) {
    return 1;
    }
    return 0;
    }
}
```

Listas: Java.Util.List

O critério de ordenação é totalmente aberto, definido pelo programador.

Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método **compareTo** na classe

Quando chamarmos o método sort de Collections, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método compareTo.

Listas: Java.Util.List

Mas, e o exemplo anterior, com uma lista de Strings? Por que a ordenação funcionou, naquele caso, sem precisarmos fazer nada?

Simples: quem escreveu a classe String (lembre que ela é uma classe como qualquer outra) implementou a interface Comparable e o método compareTo para Strings, fazendo comparação em ordem alfabética.

Consulte a documentação da classe String e veja o método compareTo lá.

O mesmo acontece com outras classes como Integer, BigDecimal, Date, entre outras.

Outros Métodos da Classe Collections

binarySearch(List, Object): Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado

max(Collection): Retorna o maior elemento da coleção.

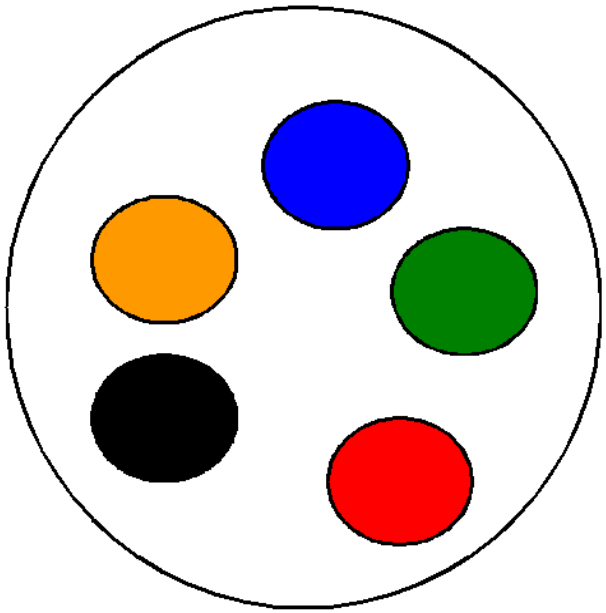
min(Collection): Retorna o menor elemento da coleção.

reverse(List): Inverte a lista.

...e muitos outros. Consulte a documentação para ver outros métodos.

Conjunto: Java.Util.Set

Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.



Possíveis ações em um conjunto:

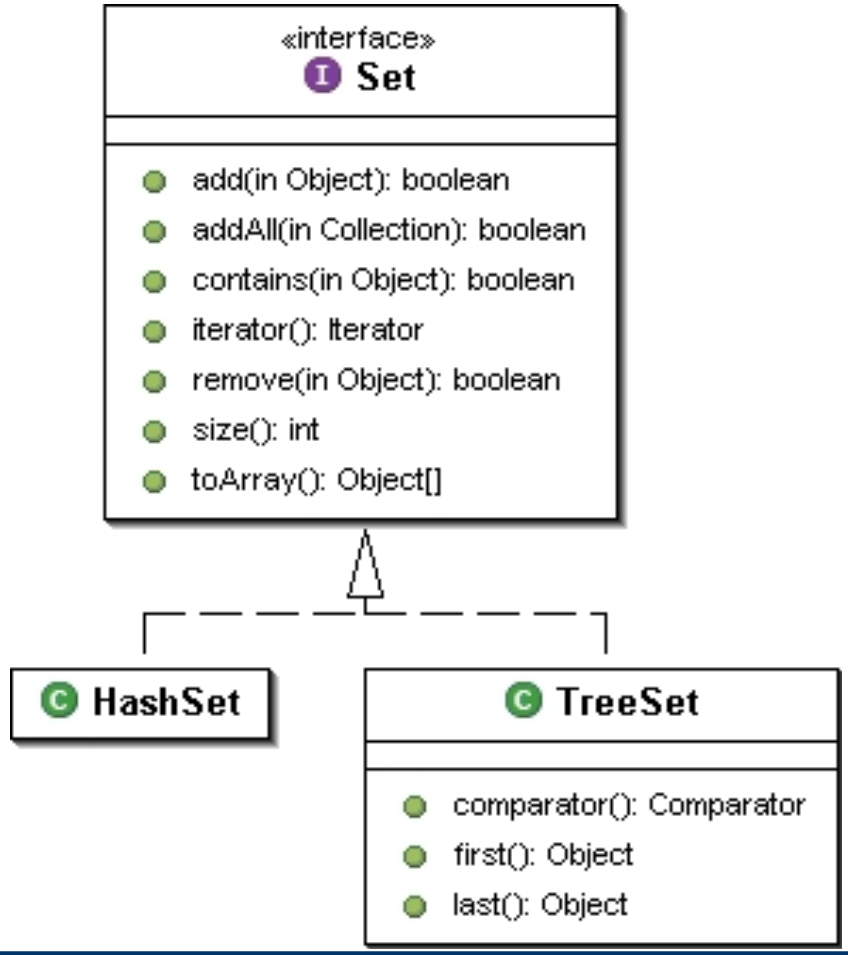
- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- Não existem elementos duplicados!

- Ao percorrer um conjunto, sua ordem não é conhecida!

Conjunto: Java.Util.Set

Um conjunto é representado pela interface Set e tem como suas principais implementações as classes HashSet e TreeSet.



Conjunto: Java.Util.Set

```
Set<String> cargos = new HashSet<>();

cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Secretária");
cargos.add("Funcionário");
cargos.add("Diretor"); // repetido!

// imprime na tela todos os elementos
System.out.println(cargos);
```

Aqui, o segundo Diretor não será adicionado e o método add lhe retornará false.

Conjunto: Java.Util.Set

O uso de um **Set** pode parecer desvantajoso:

Ele não armazena a ordem, e não aceita elementos repetidos.

Não há métodos que trabalham com índices, como o get(int) que as listas possuem.

A grande vantagem do Set é que

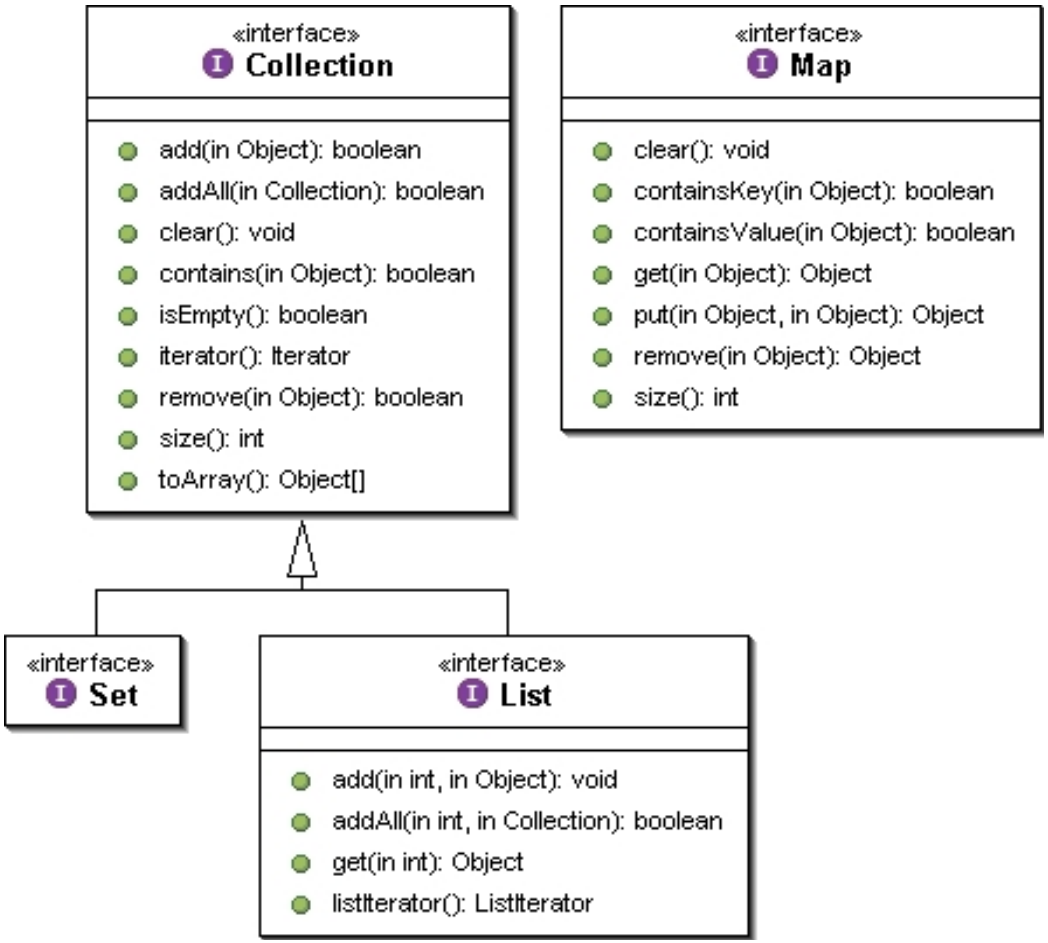
Existem implementações, como a HashSet, que possui uma performance incomparável com as Lists quando usado para pesquisa (método contains por exemplo).

Veremos essa enorme diferença em um exemplo

Principais Interfaces: Java.Util.Collection

- As coleções têm como base a interface Collection, que define métodos para adicionar e remover um elemento, e verificar se ele está na coleção, entre outras operações, como mostra a tabela a seguir:

`boolean add(Object)`	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna `true` ou `false` indicando se a adição foi efetuada com sucesso.
`boolean remove(Object)`	Remove determinado elemento da coleção. Se ele não existia, retorna `false`.
`int size()`	Retorna a quantidade de elementos existentes na coleção.
`boolean contains(Object)`	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método `equals()` do objeto, e não através do operador ==.
`Iterator iterator()`	Retorna um objeto que possibilita percorrer os elementos daquela coleção.



Uma coleção pode implementar diretamente a interface Collection, porém normalmente se usa uma das duas sub interfaces mais famosas: justamente Set e List.

Percorrendo Coleções

Como percorrer os elementos de uma coleção?

Se for uma lista, podemos sempre utilizar um laço for, invocando o método get para cada elemento.

Mas e se a coleção não permitir indexação?

Por exemplo, um Set não possui um método para pegar o primeiro, o segundo ou o quinto elemento do conjunto, já que um conjunto não possui o conceito de “ordem”

Podemos usar o enhanced-for (o “foreach”) para percorrer qualquer Collection sem nos preocupar com isso.

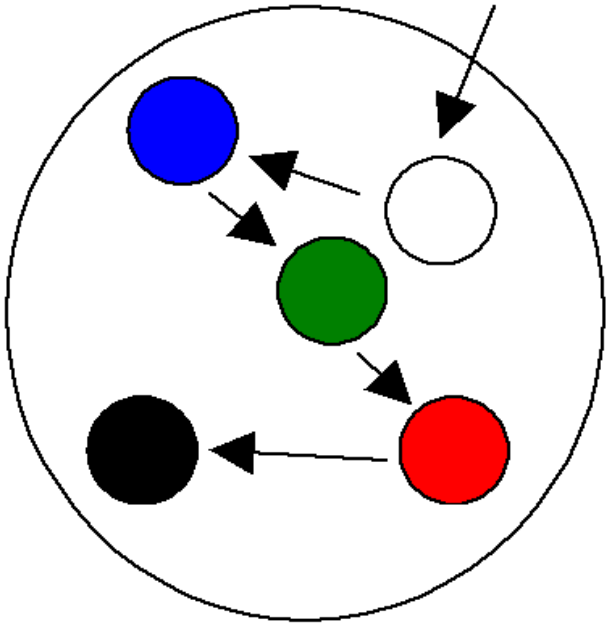
Percorrendo Coleções

```
Set<String> conjunto = new HashSet<>();
conjunto.add("java");
conjunto.add("raptor");
conjunto.add("scala");

for (String palavra : conjunto) {
   System.out.println(palavra);
}
```

- Em que ordem os elementos serão acessados?
  - Em um conjunto, a ordem depende da implementação da interface Set: você muitas vezes não vai saber ao certo em que ordem os objetos serão percorridos

Percorrendo Coleções

- Toda coleção fornece acesso a um iterator, um objeto que implementa a interface Iterator, que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura baixo mostra.



Possíveis ações em um iterador:

- Existe um próximo elemento na coleção?.
- Pegue o próximo elemento.
- Remova o elemento atual da coleção.

```
ContaCorrente c =
  (ContaCorrente) iterator.next();
```

Percorrendo Coleções

- Primeiro criamos um Iterator que entra na coleção. A cada chamada do método next, o Iterator retorna o próximo objeto do conjunto. Um iterator pode ser obtido com o método iterator() de Collection, por exemplo:

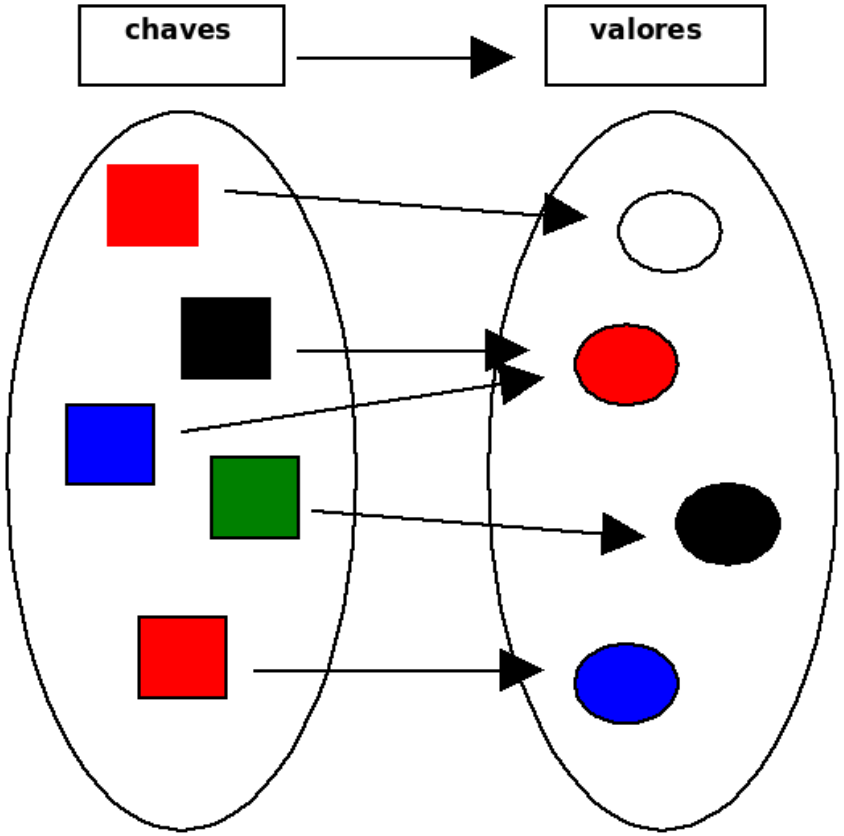
```
Set<String> conjunto = new HashSet<>();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");

Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
    // recebe a palavra
    String palavra = i.next();
    System.out.println(palavra);
}
```

Mapas – Java.Util.Maps

- Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele.
- Um exemplo seria:
  - Dada a placa do carro, obter todos os dados do carro.
  - Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes.

Mapas – Java.Util.Maps

- Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor.



Possíveis ações em um mapa:

Mapeie uma chave a um valor
O que está mapeado na chave X?
Remapeie uma certa chave
Quero o conjunto de chaves.
Quero o conjunto de valores.
Desmapeie a chave X.

Mapas – Java.Util.Maps

- O método **put(object, object)** da interface Map recebe a chave e o valor de uma nova associação.
- Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método get(Object).
- Essas são as duas operações principais e mais frequentes realizadas sobre um mapa

Mapas – Java.Util.Maps

- Criamos duas contas correntes e as colocamos em um mapa associando-as aos seus donos.

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);

// cria o mapa
Map<String, ContaCorrente> mapaDeContas = new HashMap<>();
// adiciona duas chaves e seus respectivos valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);

// qual a conta do diretor? (sem casting!)
ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Mapas – Java.Util.Maps

- Um mapa é muito usado para “indexar” objetos de acordo com determinado critério, para podermos buscar esse objetos rapidamente. Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!

- Suas principais implementações são:
  - HashMap
  - TreeMap
  - Hashtable.