Fundamentos de Java

Coleções de Dados

SoftBlue
www.softblue.com.br

Tópicos Abordados

- Arrays
- Listas
  - ArrayList
  - Generics
  - Ordenação de listas
- Conjuntos
  - HashSet, LinkedHashSet e TreeSet
  - Distinção de elementos
- Mapas
  - HashMap, TreeMap

Arrays

- Arrays são utilizados para agrupar dados de um mesmo tipo

```
int[] distancias;
distancias = new int[8];
```

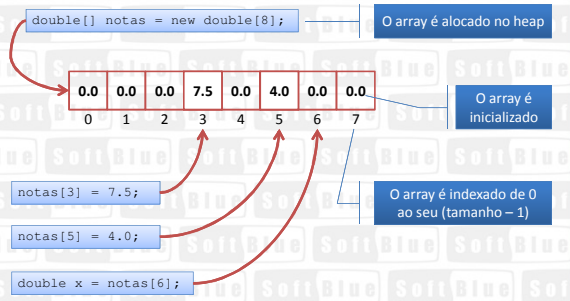
Array de **int** com **8** posições

```
double[] notas = new double[5];
```

Array de **double** com **5** posições

Em Java, array é um objeto

Acessando Elementos do Array



Considerações Sobre Arrays

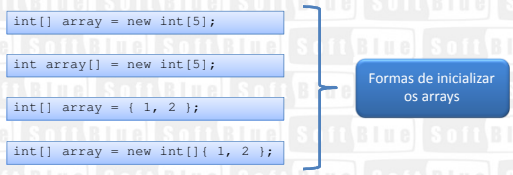
- Os índices do array vão de 0 a *n* – 1 (onde *n* é o tamanho do array)
  - Acessos fora deste intervalo resultam em erro
- Não é possível declarar arrays com tamanho negativo

```
int[] array = new int[-5];
```

- Arrays podem ter tamanho 0

```
int[] array = new int[0];
```

Inicialização de Arrays



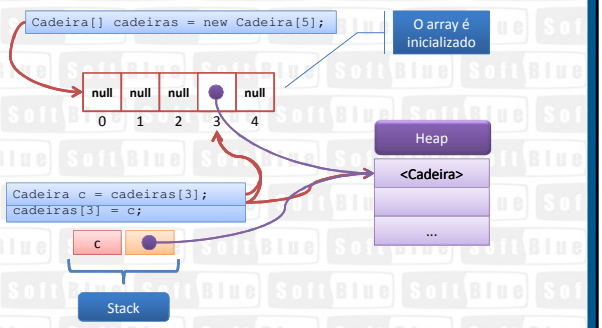
Arrays de Referências

- Além de tipos primitivos, arrays também podem guardar referências a objetos

```
Cadeira[] cadeiras = new Cadeira[5];
```

- Neste caso, cada posição do array referencia um objeto armazenado no heap

Arrays de Referências



Percorrendo Arrays

- Utilizando o *for*

```
int[] array = new int[10];

for(int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

- Utilizando o *enhanced-for*

```
int[] array = new int[10];

for(int i : array) {
    System.out.println(i);
}
```

Desvantagens dos Arrays

- Depois de criado, não é possível modificar o tamanho de um array
- Dificuldade em encontrar elementos dentro do array quando o índice não é conhecido
- Ao remover elementos, sobram “buracos” no array

A Collections API

- Possui um conjunto de classes e interfaces para facilitar o trabalho com coleções de dados
  - Listas
  - Conjuntos
  - Mapas

Listas

- Permitem elementos duplicados
- Mantêm ordenação específica entre os elementos
- Representadas pela interface **java.util.List**

Listas: ArrayList

- É a implementação de listas mais utilizada
- Trabalha internamente com um array

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

Listas: ArrayList

- Usando o método **add()**, podemos adicionar elementos no fim da lista ou em uma posição qualquer

```
List lista = new ArrayList();
lista.add("José");
lista.add("João");
lista.add(1, "Maria");
```

Listas: ArrayList

- O método **size()** retorna o tamanho da lista

```
int t = lista.size();
```

- O método **get()** retorna o elemento da posição especificada

```
Object item = lista.get(1);
```

Listas: ArrayList

- Todas as coleções são genéricas
- Trabalham apenas com tipos **Object**
- É preciso fazer *casting* da referência ao obter um elemento

```
String nome = (String) lista.get(1);
```

Percorrendo Listas

- Usando o *iterator*

```
Iterator iter = lista.iterator();

while(iter.hasNext()) {
    String nome = (String) iter.next();
    ...
}
```

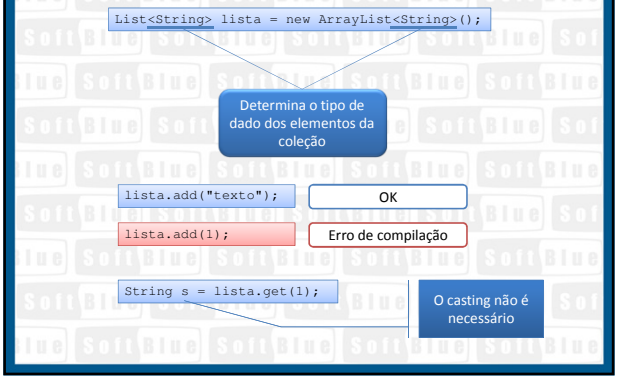
- Usando o *enhanced-for*

```
for(Object obj : lista) {
    String nome = (String) obj;
    ...
}
```

Usando Generics com Listas

- Permite restringir os tipos de dados em coleções
- Vantagens
  - Evita *casting*, que pode ser feito de forma errada
  - Faz a verificação do tipo de dado em tempo de compilação

Usando Generics com Listas



Usando Generics com Listas

- Iterar sobre listas que usam generics é mais simples

```
Iterator<String> iter = lista.iterator();

while(iter.hasNext()) {
    String nome = iter.next();
    ...
}
```

```
for(String nome : lista) {
    ...
}
```

Ordenação de Listas

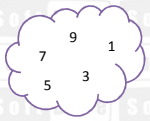
- A ordenação possibilita que os elementos fiquem posicionados de acordo com algum critério
- A classe *Collections* traz um método estático *sort()* para fazer ordenação de listas

```
Collections.sort(lista);
```

Ordenação de Listas

- A ordenação só funciona em um dos seguintes casos
  - Se os elementos da coleção implementarem a interface **java.lang.Comparable**
  - Se um **java.util.Comparator** for utilizado
- A utilização de uma dessas interfaces obriga o programador a implementar a regra de como os elementos serão ordenados

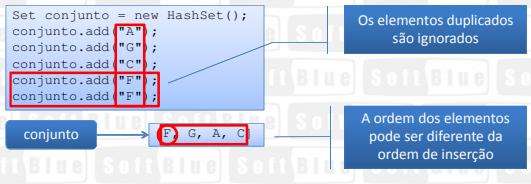
Conjuntos



- Representam conjuntos como na matemática
- Não permitem elementos duplicados
- A ordem dos elementos no conjunto pode não ser a mesma da ordem de inserção
- Representados pela interface **java.util.Set**

Conjuntos: HashSet

- Implementação de conjunto que não possui nenhuma garantia com relação à ordem dos elementos



Conjuntos: LinkedHashSet

- Garante que, ao iterar sobre os elementos, a ordem de iteração será a mesma da inserção

```
Set conjunto = new LinkedHashSet();
conjunto.add("A");
conjunto.add("G");
conjunto.add("C");
conjunto.add("F");
```

conjunto → [A, G, C, F]

A ordem dos elementos é a mesma da ordem de inserção

Conjuntos: TreeSet

- Os elementos são ordenados por algum critério no momento em que são inseridos no conjunto
- O critério é definido como nas listas
  - Implementação da interface **java.lang.Comparable**
  - Uso de um **java.util.Comparator**

Conjuntos: TreeSet

```
Set conjunto = new TreeSet();
conjunto.add("A");
conjunto.add("G");
conjunto.add("C");
conjunto.add("F");
```

conjunto → [A, C, F, G]

Os elementos são ordenados em ordem alfabética

A classe *String* implementa a interface *Comparable* e define este comportamento

Conjuntos: Distinção de Elementos

- Conjuntos não armazenam objetos iguais
  - Mas como especificar quais objetos são iguais?
- Dois métodos devem ser implementados por classes cujos objetos são usados em conjuntos
  - **equals()**
  - **hashCode()**

Percorrendo Conjuntos

- Conjuntos não são indexados
- Podem ser utilizados o *iterator* ou o *enhanced-for*

```
Iterator<String> iter = conjunto.iterator();

while(iter.hasNext()) {
    String nome = iter.next();
    ...
}
```

```
for(String nome : conjunto) {
    ...
}
```

Usando Generics com Conjuntos

- O generics também pode ser utilizado com conjuntos do mesmo modo como é feito com as listas

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

Mapas

- Utilizados quando é necessário mapear uma chave a um valor
- Chaves e valores podem ser qualquer tipo de objeto
- Representados pela interface **java.util.Map**

Mapas: HashMap

- Implementação de mapa que não possui nenhuma garantia com relação à ordem das chaves
- Os métodos **put()** e **get()** podem ser usados para adicionar e obter elementos do mapa, respectivamente

Mapas: HashMap

```
ContaCorrente c1 = new ContaCorrente(123);
ContaCorrente c2 = new ContaCorrente(321);
```

```
Map contasMap = new HashMap();
contasMap.put("cliente1", c1);
contasMap.put("cliente2", c2);
```

Associação entre clientes e contas

```
ContaCorrente c = (ContaCorrente) contasMap.get("cliente1");
```

O *get()* obtém o valor associado à determinada chave

Mapas: HashMap

- O generics também pode ser usado em mapas

```
Map<String, ContaCorrente> contasMap =
    new HashMap<String, ContaCorrente>();
```

Tipo de dado das chaves

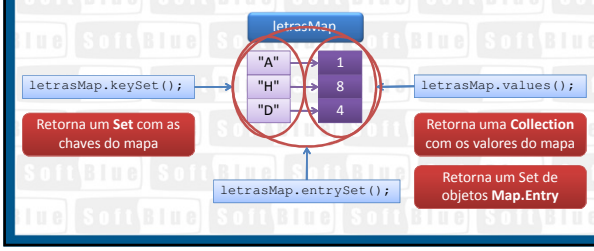
Tipo de dado dos valores

Mapas: TreeMap

- As chaves dos elementos são ordenadas por algum critério no momento em que estes são inseridos no mapa
- O critério é definido como nas listas
  - Implementação da interface **java.lang.Comparable**
  - Uso de um **java.util.Comparator**

Mapas: Retornando Coleções

- A interface **java.util.Map** possui métodos para retornar sua lista de chaves e de valores, e até cada entrada chave/valor do mapa



Inferência de Tipos em Generics

- Até o Java 6, o uso de generics deveria ser feito da seguinte forma

```
List<String> l = new ArrayList<String>();
```

- A partir do Java 7, a inferência do tipo ocorre automaticamente

```
List<String> l = new ArrayList<>();
```

Colocando em Prática...

Agora que você já aprendeu a teoria, acesse as vídeo-aulas práticas e pratique os assuntos abordados neste módulo!

Clique aqui para acessar as vídeo-aulas práticas