

Aula 21

Coleções Genéricas

Introdução

- A **biblioteca de coleções do Java** contém estruturas de dados predefinidas, interfaces e métodos para manipular essas estruturas de dados
- Com as coleções, pode-se utilizar estruturas de dados existentes, sem se preocupar com a maneira como são implementadas
- Você pode codificar mais rápido e esperar excelente desempenho, maximização da velocidade de execução e minimização do consumo de memória

Visão Geral

- Uma coleção é uma estrutura de dados que na prática é um objeto que pode armazenar referências a outros objetos
- Normalmente, as coleções contêm referências a objetos que são do mesmo tipo
- As interfaces de estruturas de coleções declaram as operações a serem realizadas (genericamente) em vários tipos de coleções

Classes Empacotadoras

- **As coleções referenciam ou armazenam objetos**
- **Para armazenar informações correspondentes à tipos primitivos, é necessário utilizar uma classe empacotadora, que permite manipular valores de tipo primitivo como objetos**
- Todo tipo primitivo possui uma classe empacotadora → Boolean, Byte, Character, Double, Float, Integer, Long e Short
- Classes empacotadoras possuem métodos e campos úteis para modificar, verificar e converter valores

Interface Collection

- A interface `Collection` é a interface-raiz na hierarquia de coleções
- Métodos como `add`, `clear`, `contains`, `equals`, `remove` e `size` estão especificados na interface `Collection`
- As interfaces `Set`, `Queue` e `List` são derivadas de `Collection`
- Além disso, há também a interface `Map`, a qual não implementa a interface `Collection` mas pertence ao *hall* de coleções do Java

Interface Collection

- A interface `Set` define uma coleção que não contém duplicatas
- A interface `Queue` define uma coleção que representa uma fila → inserções são feitas na parte de trás e exclusões na parte da frente
- A interface `List` representa uma coleção ordenada → o usuário dessa interface tem controle sobre onde os elementos serão inseridos
- A interface `Map` representa uma coleção de mapeamentos *chave* → *valor*

Listas

- Uma List (**lista** ou **sequência**) é uma Collection ordenada que pode conter elementos duplicados
- Os índices de List tem base em zero (índice do primeiro elemento é zero)
- Além dos métodos herdados de Collection, List fornece métodos para manipular elementos via seus índices, manipular um intervalo especificado de elementos, procurar elementos e obter um ListIterator para acessar os elementos
- A interface List é implementada por várias classes (ex: ArrayList, LinkedList e Vector)

Listas

- As classes `ArrayList` e `Vector` são implementações de *array* redimensionáveis de `List`
- Inserir um elemento entre elementos existentes de um `ArrayList` ou `Vector` é uma operação ineficiente → todos os elementos depois do elemento novo devem ser movidos de lugar
- As classes `ArrayList` e `Vector` têm comportamentos quase idênticos → a principal diferença entre elas é que `Vectors` são sincronizados por padrão, enquanto `ArrayLists` não o são

ArrayList e Iterator

- Em geral, as coleções não sincronizadas fornecem melhor desempenho que as sincronizadas
- Um `LinkedList` permite inserção (ou remoção) eficiente de elementos no meio de uma coleção
- Os principais métodos da classe `ArrayList` (e em geral das classes que implementam `Collection`) são:
 - `add`: adiciona um objeto na lista
 - `contains`: verifica se a lista contém um determinado elemento
 - `remove`: remove um objeto da lista
 - `get`: retorna um objeto da lista
 - `size`: retorna o tamanho da lista
 - `iterator`: retorna um `Iterator` para a lista

ArrayList e Iterator

- Com um Iterator é possível percorrer a lista e até mesmo remover elementos da lista
- Os principais métodos de um Iterator são:
 - hasNext: retorna verdadeiro ainda existem mais elementos a serem percorridos na lista
 - next: retorna o próximo elemento da iteração
 - remove: remove o último elemento retornado pelo Iterator

ArrayList e Iterator

```
17 public class TesteCollections {  
18  
19     public static void main(String[] args){  
20  
21         ArrayList<String> listaCores = new ArrayList<String>();  
22         listaCores.add("Vermelho"); listaCores.add("Azul");  
23         listaCores.add("Amarelo"); listaCores.add("Verde");  
24  
25         ArrayList<String> listaRemover = new ArrayList<String>();  
26         listaRemover.add("Vermelho"); listaRemover.add("Amarelo");  
27  
28         removerCores(listaCores, listaRemover);  
29  
30         System.out.println("Lista de cores final =====");  
31         for(String str : listaCores){  
32             System.out.println(str);  
33         }  
34     }  
35  
36     public static void removerCores(Collection<String> lista1, Collection<String> lista2){  
37         Iterator<String> iterator = lista1.iterator();  
38         while(iterator.hasNext()){  
39             if(lista2.contains(iterator.next())){  
40                 iterator.remove();  
41             }  
42         }  
43     }  
44  
45 }  
46  
47 }
```

Localizar: loadPro | Anterior | Próximo |

Resultados da Pesquisa | Salva x

Console do Depurador x | Teste (run) x

run:
Lista de cores final =====
Azul
Verde

ArrayList e Iterator

- Vale ressaltar que para o correto funcionamento do método `contains(...)`, deve-se implementar o método `equals(...)` das classes que não tem tal método implementado
- Caso o método `equals` não seja implementado, dois objetos de mesmo tipo e com o mesmos valores dos atributos serão considerados objetos diferentes

```
16 public class TesteArray {  
17  
18     public static void main(String[] args) {  
19  
20         ArrayList<ContaBancaria> contas = new ArrayList<ContaBancaria>();  
21  
22         ContaBancaria conta1 = new ContaBancaria("Rafael", 0, 250.0);  
23         ContaBancaria conta2 = new ContaBancaria("Rafael", 0, 250.0);  
24         ContaBancaria conta3 = new ContaBancaria("Ricardo", 1, 350.0);  
25  
26         contas.add(conta1);  
27         contas.add(conta3);  
28         System.out.println(contas.contains(conta2));  
29     }  
30 }  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```

ArrayList

```
16 public class ContaBancaria{
17
18     private String proprietario;
19     private int id;
20     private double saldo;
21
22     public ContaBancaria(String proprietario, int id, double saldo) {
23         this.proprietario = proprietario;
24         this.id = id;
25         this.saldo = saldo;
26     }
27
28     @Override
29     public boolean equals(Object obj) {
30         ContaBancaria outraConta = (ContaBancaria)obj;
31         if(this.proprietario.equals(outraConta.proprietario) &&
32             this.id == outraConta.id && this.saldo == outraConta.saldo){
33             return true;
34         }else{
35             return false;
36         }
37     }
38 }
```

```
16 public class TesteArray {
17
18     public static void main(String[] args) {
19
20         ArrayList<ContaBancaria> contas = new ArrayList<ContaBancaria>();
21
22         ContaBancaria conta1 = new ContaBancaria("Rafael", 0, 250.0);
23         ContaBancaria conta2 = new ContaBancaria("Rafael", 0, 250.0);
24         ContaBancaria conta3 = new ContaBancaria("Ricardo", 1, 350.0);
25
26         contas.add(conta1);
27         contas.add(conta3);
28         System.out.println(contas.contains(conta2));
29     }
30 }
31
32
33
34
35
36
37
```

Saida - Teste (run) x Resultados da Pesquisa x

```
run:
true
```

LinkedList

- Além dos métodos de um ArrayList, uma LinkedList possui métodos para inserir no começo e no final da lista respectivamente com os métodos addLast e addFirst
- Tanto um ArrayList quanto um LinkedList permite retornar um ListIterator, o qual é um iterador bidimensional e permite recuperar, alterar, adicionar e remover valores em uma lista

LinkedList

```
20 public class TesteCollections {
21
22     public static void main(String[] args){
23         ArrayList<String> testal = new ArrayList<String>();
24
25         LinkedList<String> lista1 = new LinkedList<String>();
26         lista1.add("vermelho"); lista1.add("azul"); lista1.add("amarelo");
27
28         LinkedList<String> lista2 = new LinkedList<String>();
29         lista2.add("verde"); lista2.add("roxo"); lista2.add("laranja");
30
31         lista1.addAll(lista2);
32         lista2 = null;
33
34         System.out.println("Imprimindo Lista...");
35         imprimirLista(lista1);
36
37         System.out.println("Imprimindo Sublista...");
38         imprimirLista(lista1.subList(0, 3));
39     }
40
41     public static void imprimirLista(List<String> lista){
42         ListIterator iterator = lista.listIterator(lista.size());
43         while(iterator.hasPrevious()){
44             System.out.println(iterator.previous());
45         }
46     }
47
48 }
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

run:
Imprimindo Lista...
laranja
roxo
verde
amarelo
azul
vermelho
Imprimindo Sublista...
amarelo
azul
vermelho

De Lists para Arrays

- Listas e outras coleções podem ser convertidas para arrays utilizando o método `toArray`

```
20 public class TesteCollections {
21
22     public static void main(String[] args){
23         ArrayList<String> testel = new ArrayList<String>();
24         testel.add("rosa"); testel.add("roxo"); testel.add("pink"); testel.add("links");
25
26         String[] arrayCores = testel.toArray(new String[testel.size()]);
27
28         System.out.println("Imprimindo ArrayCores -----");
29         for(String str : arrayCores){
30             System.out.println(str);
31         }
32     }
33 }
34
35
36
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

```
run:
Imprimindo ArrayCores -----
rosa
roxo
pink
links
```


Classe Collections

- A classe Collections fornece vários algoritmos de alto desempenho para manipular elementos de coleções
- Os algoritmos são implementados como métodos static

Método	Descrição
sort	Classifica os elementos de uma List.
binarySearch	Localiza um objeto em uma List.
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.

Classe Collections

- Os métodos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operam em `Lists`
- Os métodos `min`, `max`, `addAll`, `frequency` e `disjount` operam em `Collections`

Classe Collections - método sort

```
16 public class TesteCollections {
17
18     public static void main(String[] args){
19
20         ArrayList<String> testel = new ArrayList<String>();
21         testel.add("Programação");
22         testel.add("Orientada");
23         testel.add("Objetos");
24
25         System.out.println("Shuffle =====");
26         Collections.shuffle(testel);
27         Iterator<String> iterator = testel.iterator();
28         while(iterator.hasNext()){
29             System.out.println(iterator.next());
30         }
31
32         System.out.println("Sort =====");
33         Collections.sort(testel);
34         iterator = testel.iterator();
35         while(iterator.hasNext()){
36             System.out.println(iterator.next());
37         }
38
39         System.out.println("ReplaceAll =====");
40         Collections.replaceAll(testel, "Objetos", "Objeto");
41         iterator = testel.iterator();
42         while(iterator.hasNext()){
43             System.out.println(iterator.next());
44         }
45     }
46 }
47
48 }
```

Localizar: loadPro

TesteCollections main

Resultados da Pesquisa

Console do Depurador x Teste (run) x

run:
Shuffle =====
Programação
Objetos
Orientada
Sort =====
Objetos
Orientada
Programação
ReplaceAll =====
Objeto
Orientada
Programação

Classe Collections - método sort

- Pode-se definir também um `Comparator`, que dirá se um objeto é “menor” ou “maior” que o outro
- Isso é muito útil para comparar objetos com mais de um campo
- Para se implementar um `Comparator`, deve-se implementar a interface `Comparator` e implementar o método `compare(Object objeto1, Object objeto2)`
- Se o método `Comparator` retornar
 - **Um valor positivo**: significa que o `objeto1` é maior que o `objeto2`
 - **Um valor negativo**: significa que o `objeto1` é menor que o `objeto2`
 - **Valor igual a 0**: significa que `objeto1` é igual ao `objeto2`

Classe Collections - método sort

```
12 public class Hora {  
13  
14     private int hora;  
15     private int minuto;  
16     private int segundo;  
17  
18     public Hora(int hora, int minuto, int segundo) {  
19         this.hora = hora;  
20         this.minuto = minuto;  
21         this.segundo = segundo;  
22     }  
23  
24     public int getHora() {  
25         return hora;  
26     }  
27  
28     public void setHora(int hora) {  
29         this.hora = hora;  
30     }  
31  
32     public int getMinuto() {  
33         return minuto;  
34     }  
35  
36     public void setMinuto(int minuto) {  
37         this.minuto = minuto;  
38     }  
39  
40     public int getSegundo() {  
41         return segundo;  
42     }  
43  
44     public void setSegundo(int segundo) {  
45         this.segundo = segundo;  
46     }  
47  
48 }
```

Classe Collections - método sort

```
14 public class ComparadorHora implements Comparator<Hora>{
15
16     @Override
17     public int compare(Hora hora1, Hora hora2) {
18
19         int difHora = hora1.getHora() - hora2.getHora();
20         if(difHora != 0){
21             return difHora;
22         }
23
24         int difMin = hora1.getMinuto() - hora2.getMinuto();
25         if(difMin != 0){
26             return difMin;
27         }
28
29         int difSeg = hora1.getSegundo() - hora2.getSegundo();
30         return difSeg;
31     }
32 }
33
34 }
```

Classe Collections - método sort

```
17 public class TesteComparator {  
18  
19     public static void main(String[] args){  
20  
21         ArrayList<Hora> horas = new ArrayList<Hora>();  
22  
23         horas.add(new Hora(23, 15, 30));  
24         horas.add(new Hora(7, 50, 11));  
25         horas.add(new Hora(14, 30, 45));  
26  
27  
28         Collections.sort(horas, new ComparadorHora());  
29  
30         for(Hora hora : horas){  
31             System.out.println(hora.toString());  
32         }  
33     }  
34 }  
35  
36
```

Resultados da Pesquisa

Saída x



Console do Depurador x

Teste (run) x



```
run:  
23:15:30  
14:30:45  
7:50:11
```

Pilhas

- A classe Stack estende a classe Vector para implementar uma estrutura de dados de pilha
- Devem ser utilizados os elementos push e pop para adicionar e obter/remover elementos de uma pilha respectivamente
- Outros métodos da classe Vector podem ser utilizados, porém, não há garantias de manter o comportamento de uma pilha

Pilhas

```
16 public class TesteComparator {
17
18     public static void main(String[] args){
19
20         Stack<Integer> pilha = new Stack<Integer>();
21
22         pilha.push(0);
23         pilha.push(5);
24         pilha.push(10);
25         pilha.push(15);
26         pilha.push(20);
27
28
29         System.out.println("Desempilhando elementos:");
30         while(pilha.size() > 0){
31             System.out.println(pilha.pop());
32         }
33     }
34 }
35
```

Fila

- Em uma fila, as inserções são feitas na parte posterior da fila e as exclusões na parte da frente
- Em Java, têm-se a interface Queue e a classe PriorityQueue
- A interface Queue estende a interface Collection e fornece operações adicionais para inserir, remover e inspecionar elementos em uma fila

Fila

- `PriorityQueue` implementa a interface `Queue`, ordena elementos por sua ordem natural como especificado pelo método `compareTo` dos elementos `Comparable` ou por um objeto `Comparator` que é fornecido pelo construtor
- A classe `PriorityQueue` fornece funcionalidades que permitem inserções na ordem de classificação na estrutura de dados subjacente e exclusões a partir da frente da estrutura de dados subjacente

Fila

- Ao adicionar elementos a uma `PriorityQueue`, os elementos são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade será o primeiro elemento removido da `PriorityQueue`
- As operações `PriorityQueue` comuns são `offer`, para inserir um elemento na posição apropriada com base na ordem de prioridade, `poll`, para remover o elemento de mais alta prioridade da fila de prioridade (isto é, a cabeça da fila), `peek` para obter uma referência ao elemento de mais alta prioridade da fila de prioridade

PriorityQueue

```
16 public class Fila {  
17  
18     public static void main(String[] args){  
19  
20         PriorityQueue<Double> filaP = new PriorityQueue<Double>();  
21  
22         filaP.offer(3.2);  
23         filaP.offer(9.7);  
24         filaP.offer(1.5);  
25  
26         System.out.println("Imprimindo e removendo elementos da fila: ");  
27         while(filaP.size() > 0){  
28             System.out.println(filaP.poll());  
29         }  
30     }  
31 }  
32
```

Resultados da Pesquisa

Saída x



Console do Depurador x

Teste (run) x



```
run:  
Imprimindo e removendo elementos da fila:  
1.5  
3.2  
9.7
```

PriorityQueue

```
16 public class Fila {  
17  
18     public static void main(String[] args){  
19  
20         PriorityQueue<Hora> filaP = new PriorityQueue<Hora>(1, new ComparadorHora());  
21  
22         filaP.offer(new Hora(21, 7, 36));  
23         filaP.offer(new Hora(7, 9, 35));  
24         filaP.offer(new Hora(14, 35, 24));  
25  
26         System.out.println("Imprimindo e removendo elementos da fila: ");  
27         while(filaP.size() > 0){  
28             System.out.println(filaP.poll());  
29         }  
30  
31     }  
32 }
```

Resultados da Pesquisa

Saída x



Console do Depurador x

Teste (run) x



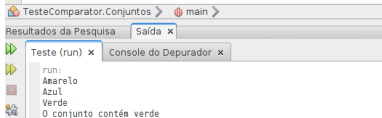
```
run:  
Imprimindo e removendo elementos da fila:  
7:9:35  
14:35:24  
21:7:36
```

Conjuntos

- Um Set é um Collection de elementos únicos → sem elementos duplicados
- A estrutura de coleções contém diversas implementações de Set, incluindo HashSet e TreeSet
- HashSet armazena seus elementos em uma tabela de Hash (não ordenado)
- TreeSet armazena seus elementos em uma árvore rubro negra de maneira ordenada

HashSet

```
16 public class Conjuntos {  
17  
18     public static void main(String[] args){  
19  
20         HashSet<String> setCores = new HashSet<String>();  
21  
22         setCores.add("Azul");  
23         setCores.add("Amarelo");  
24         setCores.add("Azul");  
25         setCores.add("Amarelo");  
26         setCores.add("Verde");  
27  
28         for(String str : setCores){  
29             System.out.println(str);  
30         }  
31  
32         if(setCores.contains("Verde")){  
33             System.out.println("O conjunto contém verde");  
34         }  
35     }  
36 }  
37  
38 }  
39
```



HashSet

- Lembrando da teoria do tabela hash, é necessário gerar um valor inteiro para um elemento para descobrir qual será o índice em que este será alocado na tabela

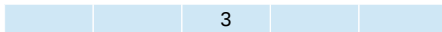
Tabela Hash de tamanho 5



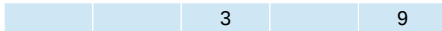
Função Hash → $f(x) = x \bmod \text{tamanho_tabela_hash}$



Elemento: **3**



Elemento: **9**



HashSet

- Porém, dado um elemento do conjunto que não seja um inteiro, por exemplo, uma `String`, `ContaBancária` ou `Pessoa`, é necessário definir como gerar o valor inteiro para tal elemento (objeto) para que este valor inteiro seja submetido à uma função de hash
- Todo objeto no java possui um método `hashCode()` que retorna um valor inteiro (método da classe `Object`)
- Porém, sua versão implementada na classe `Object` pode gerar valores inteiros diferentes para objetos que contenham os mesmos valores de atributos

HashSet

- **OBSERVAÇÃO:** classe como String e os Wrapper já sobrescrevem o método hashCode de forma a gerar o mesmo valor para objetos diferentes

```
15 public class TesteHash2 {  
16  
17     public static void main(String[] args) {  
18  
19         String str1 = new String("Rafael");  
20         String str2 = new String("Rafael");  
21         String str3 = new String("Programação");  
22  
23         System.out.println(str1.hashCode());  
24         System.out.println(str2.hashCode());  
25         System.out.println(str3.hashCode());  
26  
27     }  
28 }  
29  
30 }
```

Salida - Teste (run) x Resultados da Pesquisa x

run:
-1854660239
-1854660239
-1616020618

HashSet

- Porém, para uma classe implementada na qual o método `hashCode` não foi sobrescrito...

```
public class ContaBancaria {  
    private String proprietario;  
    private int id;  
    private double saldo;  
  
    public ContaBancaria(String proprietario, int id, double saldo) {  
        this.proprietario = proprietario;  
        this.id = id;  
        this.saldo = saldo;  
    }  
}
```

```
11 public class TesteHash {  
12  
13     public static void main(String[] args) {  
14         HashSet<ContaBancaria> hash = new HashSet<ContaBancaria>();  
15  
16  
17         ContaBancaria conta1 = new ContaBancaria("Rafael", 0, 250.0);  
18         ContaBancaria conta2 = new ContaBancaria("Rafael", 0, 250.0);  
19  
20         System.out.println(conta1.hashCode());  
21         System.out.println(conta2.hashCode());  
22     }  
23  
24 }  
25
```

Saida - Teste (run) | Resultados da Pesquisa | ContaBancaria.java

```
run:  
366712642  
1829164788  
2018699554
```

HashSet

- Assim como para comparar objetos de acordo com os valores de seus campos era necessário sobrescrever o método `equals(...)`, para fazer que dois objetos com mesmos valores de atributos gerem o mesmo código hash, é necessário implementar o método `hashCode(...)`

```
public class ContaBancaria {  
    private String proprietario;  
    private int id;  
    private double saldo;  
  
    public ContaBancaria(String proprietario, int id, double saldo) {  
        this.proprietario = proprietario;  
        this.id = id;  
        this.saldo = saldo;  
    }  
  
    @Override  
    public int hashCode() {  
        int hash = this.proprietario.hashCode();  
        hash += id;  
        return hash;  
    }  
}
```

```
5 public class TesteHash2 {  
6  
7     public static void main(String[] args) {  
8  
9         ContaBancaria conta1 = new ContaBancaria("Rafael", 0, 250.0);  
10        ContaBancaria conta2 = new ContaBancaria("Rafael", 0, 250.0);  
11        ContaBancaria conta3 = new ContaBancaria("Ricardo", 1, 350.0);  
12  
13        System.out.println(conta1.hashCode());  
14        System.out.println(conta2.hashCode());  
15        System.out.println(conta3.hashCode());  
16  
17    }  
18  
19 }  
20
```

Salda - Teste (run) X Resultados da Pesquisa X

RUN:
-1854668239
-1854668239
-1433617959

HashSet

- Porém, implementar somente o `hashCode(...)` não é garantia de sucesso para a implementação de conjuntos, uma vez que diferentes elementos podem gerar o mesmo valor da função *hash*
- É necessário também sobrescrever o método `equal(...)` para que caso dois elementos gerem o mesmo *hash*, eles sejam armazenados caso sejam diferentes, e descartados caso sejam iguais

HashSet

Tabela Hash de tamanho 5



Função Hash → $f(x) = x \bmod \text{tamanho_tabela_hash}$



Elemento: 3



Elemento: 7



HashSet

HashSet com colisões e sem a implementação do método equals(...)

```
public class ContaBancaria {  
    private String proprietario;  
    private int id;  
    private double saldo;  
  
    public ContaBancaria(String proprietario, int id, double saldo) {  
        this.proprietario = proprietario;  
        this.id = id;  
        this.saldo = saldo;  
    }  
  
    @Override  
    public int hashCode() {  
        // Apenas para forçar a colisão  
        int hash = 7;  
        return hash;  
    }  
  
    @Override  
    public String toString() {  
        return "ContaBancaria{" + "proprietario=" + proprietario + ", id=" + id + ", saldo=" + saldo + "}";  
    }  
}
```

```
public class Aula21 {  
  
    public static void main(String[] args) {  
  
        HashSet<ContaBancaria> contas = new HashSet<>();  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
  
        for(ContaBancaria conta : contas){  
            System.out.println(conta);  
        }  
    }  
}
```



```
ContaBancaria{proprietario=Rafael, id=1, saldo=200.1}  
ContaBancaria{proprietario=Rafael, id=1, saldo=200.1}  
ContaBancaria{proprietario=Rafael, id=1, saldo=200.1}
```


HashSet

HashSet com colisões e sem a implementação do método equals(...)

```
public class ContaBancaria {  
    private String proprietario;  
    private int id;  
    private double saldo;  
  
    public ContaBancaria(String proprietario, int id, double saldo) {  
        this.proprietario = proprietario;  
        this.id = id;  
        this.saldo = saldo;  
    }  
  
    @Override  
    public int hashCode() {  
        // Apenas para forçar a colisão  
        int hash = 7;  
        return hash;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        final ContaBancaria other = (ContaBancaria) obj;  
        if (this.id != other.id) { return false; }  
        if (this.saldo != other.saldo) { return false; }  
        if (!this.proprietario.equals(other.proprietario)) { return false; }  
        return true;  
    }  
  
    @Override  
    public String toString() {  
        return "ContaBancaria{" + "proprietario=" + proprietario + ", id=" + id + ", saldo=" + saldo + "}";  
    }  
}
```



```
public class Aula21 {  
  
    public static void main(String[] args) {  
  
        HashSet<ContaBancaria> contas = new HashSet<>();  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
        contas.add(new ContaBancaria("Rafael", 1, 200.1));  
  
        for(ContaBancaria conta : contas){  
            System.out.println(conta);  
        }  
    }  
}
```

ContaBancaria{proprietario=Rafael, id=1, saldo=200.1}

TreeSet

- A interface SortedSet mantém elementos de conjuntos de maneira ordenada, considerando a ordem natural dos elementos ou uma ordem especificada por um Comparator
- A classe TreeSet implementa um SortedSet

TreeSet

```
16 public class Conjuntos3 {  
17  
18     public static void main(String[] args){  
19  
20         SortedSet<String> setCores = new TreeSet<String>();  
21  
22         setCores.add("Verde");  
23         setCores.add("Azul");  
24         setCores.add("Amarelo");  
25         setCores.add("Azul");  
26         setCores.add("Amarelo");  
27  
28         for(String str : setCores){  
29             System.out.println(str);  
30         }  
31  
32         if(setCores.contains("Verde")){  
33             System.out.println("O conjunto contém verde");  
34         }  
35  
36     }  
37  
38 }  
39
```

Resultados da Pesquisa

Saída x

run:
Amarelo
Azul
Verde
O conjunto contém verde

TreeSet

```
16 public class Conjuntos3 {
17
18     public static void main(String[] args){
19
20         SortedSet<String> setCores = new TreeSet<String>();
21
22         setCores.add("Verde");
23         setCores.add("Azul");
24         setCores.add("Amarelo");
25         setCores.add("Azul");
26         setCores.add("Amarelo");
27
28         for(String str : setCores){
29             System.out.println(str);
30         }
31
32         if(setCores.contains("Verde")){
33             System.out.println("O conjunto contém verde");
34         }
35
36         System.out.println("Primeiro elemento do conjunto: " + setCores.first());
37         System.out.println("Último elemento do conjunto: " + setCores.last());
38     }
39 }
40
41
42 }
```

Resultados da Pesquisa Salda x

Console do Depurador x Teste (run) x Teste (run) #2 x

run:
Amarelo
Azul
Verde
O conjunto contém verde
Primeiro elemento do conjunto: Amarelo
Último elemento do conjunto: Verde

TreeSet

```
16 public class Conjuntos3 {
17
18     public static void main(String[] args){
19
20         SortedSet<String> setCores = new TreeSet<String>();
21
22         setCores.add("Verde");
23         setCores.add("Azul");
24         setCores.add("Amarelo");
25         setCores.add("Azul");
26         setCores.add("Amarelo");
27         setCores.add("Roxo");
28         setCores.add("Abóbora");
29         setCores.add("Pink");
30
31         System.out.println("Imprimindo o conjunto completo");
32         for(String str : setCores){
33             System.out.println(str);
34         }
35
36         System.out.println("Imprimindo o headSet de Azul");
37         for(String str : setCores.headSet("Azul")){
38             System.out.println(str);
39         }
40
41         System.out.println("Imprimindo o tailSet de Azul");
42         for(String str : setCores.tailSet("Azul")){
43             System.out.println(str);
44         }
45     }
46 }
47
48 }
```

TesteComparator.Conjuntos3 > main > for (String str : setCores.tailSet("Azul"))

Resultados da Pesquisa x Saída x

Console do Depurador x Teste (run) x Teste (run) #2 x

run:
Imprimindo o conjunto completo
Abóbora
Amarelo
Azul
Pink
Roxo
Verde
Imprimindo o headSet de Azul
Abóbora
Amarelo
Imprimindo o tailSet de Azul
Azul
Pink
Roxo
Verde

TreeSet

- Não implementar a interface Comparable irá causar erro, pois o TreeSet necessita saber como ordenar os dados

```
14 public class TesteTreeSet {  
15  
16     public static void main(String[] args) {  
17  
18         TreeSet<ContaBancaria> tree = new TreeSet<ContaBancaria>();  
19  
20         tree.add(new ContaBancaria("Rafael", 0, 250.0));  
21         tree.add(new ContaBancaria("Abelardo", 2, 550.0));  
22         tree.add(new ContaBancaria("Ricardo", 4, 350.0));  
23  
24  
25         for(ContaBancaria conta : tree){  
26             System.out.println(conta.toString());  
27         }  
28     }  
29 }
```

run:
Exception in thread "main" java.lang.ClassCastException: ContaBancaria cannot be cast to java.lang.Comparable
at java.util.TreeMap.compare(TreeMap.java:1290)
at java.util.TreeMap.put(TreeMap.java:538)
at java.util.TreeSet.add(TreeSet.java:255)
at TesteTreeSet.main(TesteTreeSet.java:20)

TreeSet

```
13 public class ContaBancaria implements Comparable<ContaBancaria>{
14
15     private String proprietario;
16     private int id;
17     private double saldo;
18
19     public ContaBancaria(String proprietario, int id, double saldo) {
20         this.proprietario = proprietario;
21         this.id = id;
22         this.saldo = saldo;
23     }
24
25     @Override
26     public int compareTo(ContaBancaria o) {
27         return this.proprietario.compareTo(o.proprietario);
28     }
29 }
```

```
14 public class TesteTreeSet {
15
16     public static void main(String[] args) {
17
18         TreeSet<ContaBancaria> tree = new TreeSet<ContaBancaria>();
19
20         tree.add(new ContaBancaria("Rafael", 0, 250.0));
21         tree.add(new ContaBancaria("Abelardo", 2, 550.0));
22         tree.add(new ContaBancaria("Ricardo", 4, 350.0));
23
24         for(ContaBancaria conta : tree){
25             System.out.println(conta.toString());
26         }
27     }
28 }
```

TesteTreeSet

Saída - Teste (nrun) | Resultados da Pesquisa

run:

- ContaBancaria[propietario=Abelardo, id=2, saldo=550.0]
- ContaBancaria[propietario=Rafael, id=0, saldo=250.0]
- ContaBancaria[propietario=Ricardo, id=4, saldo=350.0]

Mapas

- Os mapas (MAPs) associam chaves a valores
- As chaves em um Map devem ser únicas, mas não os valores associados
- Se um Map contiver tanto chaves como valores únicos, diz-se que ele implementa um mapeamento um para um
- Se apenas as chaves forem únicas, diz-se que o Map implementa um mapeamento de muitos para um

Mapas

- Três das várias classes que implementam a interface Map são HashTable, HashMap e TreeMap
- HashTables e HashMaps armazenam elementos em tabelas de Hash e TreeMap armazenam elementos em árvores
- Assim como nos conjuntos, HashTables e HashMaps não mantêm as chaves ordenadas enquanto que o TreeMap mantém as chaves ordenadas

HashMap

- Lembrando...
 - Técnicas de *hashing* → conversão de uma chave em um valor inteiro (índice do array) → armazenar e recuperar elementos do *hash*
 - Não se pode armazenar dois valores no mesmo espaço → encontrar o próxima posição válida
 - Uma outra solução popular é fazer cada célula ser um “*bucket*” de *hash* → uma lista vinculada de todos os pares chave/valor que sofrem *hash* para essa célula

HashMap

- A principal diferença entre as classes `HashMap` e `HashTable` é que o `HashMap` é não sincronizado e permite chaves `null` e valores `null`
- O **fator de carga** de uma tabela de *hash* afeta o desempenho de esquemas de *hashing*
- O fator de carga é a relação do número de células ocupadas na tabela de *hash* com o número total de células na tabela de *hash*

HashMap

- Quanto mais a proporção se aproximar de 1, maior a chance de colisões
- Ao criar um HashMap, por padrão, este é criado com capacidade inicial igual a 16 elementos e um fator de carga 0.75
- Quando o número de posições ocupadas no HashMap tornar-se maior que a capacidade vezes o fator de carga, a capacidade é automaticamente dobrada

HashMap

- Capacidade inicial e fator de balanceamento podem ser definidos via construtores da classe
 - `HashMap(int initialCapacity)`
 - `HashMap(int initialCapacity, float loadFactor)`
- Principais métodos de um HashMap
 - `get`: retorna o valor dada uma chave
 - `put`: associa uma chave a um valor
 - `containsKey`: verifica se uma chave existe em um mapa

HashMap

```
17 public class Mapas {
18     public static void main(String[] args){
19
20         HashMap<String,String> hash = new HashMap<String,String>();
21
22         hash.put("346.047.879-10", "Rafael");
23         hash.put("847.987.987-98", "Ricardo");
24         hash.put("874.528.986-36", "Ronaldo");
25
26         System.out.println("O hash contém a chave 358.587.987-99: " + hash.containsKey("358.587.987-99"));
27         System.out.println("Valor da chave 346.047.879-10: " + hash.get("346.047.879-10"));
28
29         System.out.println("Chaves do Hash:");
30         Set<String> chaves = hash.keySet();
31         String[] arrayChaves = chaves.toArray(new String[chaves.size()]);
32         for(String ch : arrayChaves){
33             System.out.println(ch);
34         }
35
36         System.out.println("Valores do Hash:");
37         Collection<String> valores = hash.values();
38         String[] arrayValores = valores.toArray(new String[valores.size()]);
39         for(String ch : arrayValores){
40             System.out.println(ch);
41         }
42     }
43 }
44
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

RUN:
O hash contém a chave 358.587.987-99: false
Valor da chave 346.047.879-10: Rafael
Chaves do Hash:
847.987.987-98
874.528.986-36
346.047.879-10
Valores do Hash:
Ricardo
Ronaldo
Rafael

TreeMap

- O TreeMap terá funcionamento semelhante ao HashMap, porém, as chaves são armazenadas em uma árvore rubro-negra (complexidade de operações = $O(\log n)$)

```
16 public class TesteTreeMap {
17
18     public static void main(String[] args) {
19
20         TreeMap<String, String> tree = new TreeMap<String, String>();
21
22         tree.put("346.098.000-10", "Rafael");
23         tree.put("123.456.789-88", "Ricardo");
24         tree.put("548.987.654-21", "Ronaldo");
25
26         Set<String> keys = tree.keySet();
27         for(String s : keys){
28             System.out.println(s + " - " + tree.get(s));
29         }
30     }
31 }
```

run:
123.456.789-88 - Ricardo
346.098.000-10 - Rafael
548.987.654-21 - Ronaldo

Material Complementar

- Collections framework

<https://www.caelum.com.br/apostila-java-orientacao-objetos/collections-framework/>

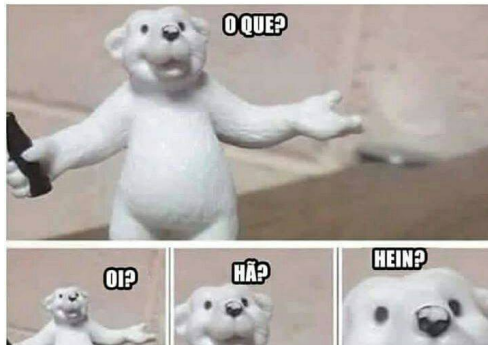
- Collections Java https://www.youtube.com/watch?v=m6JNiSY8rrE&list=PL6uyk66_MEGfrzkxhUGkQZmy3d5zRSqDS

- API Collections em Java: fundamentos e implementação básica

<http://www.devmedia.com.br/api-collections-em-java-fundamentos-e-implementacao-basica/28445>

Imagem do Dia

Quando meu código compila sem erro



Programação Orientada a Objetos

<http://lives.ufms.br/moodle/>

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br

Slides baseados em [Deitel and Deitel, 2010]

Referências Bibliográficas I



Deitel, P. and Deitel, H. (2010).

Java: How to Program.

How to program series. Pearson Prentice Hall, 8th edition.