Maps (Mapas)

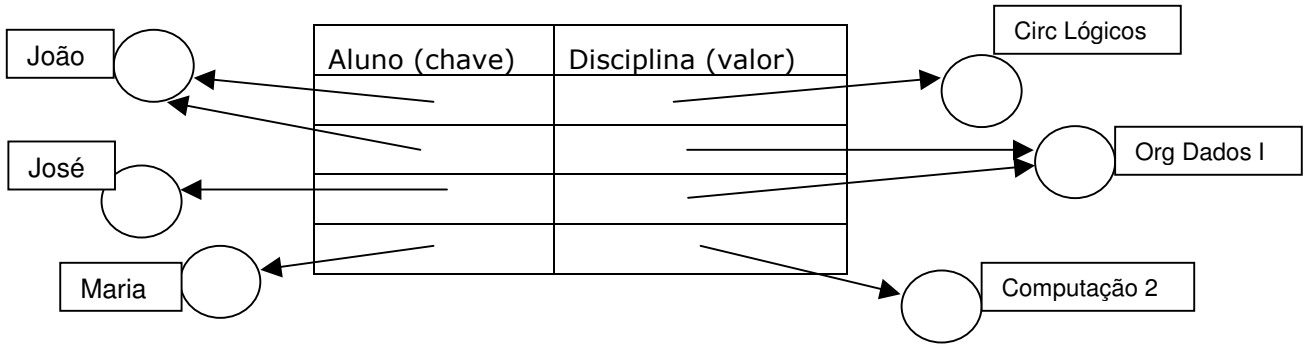
versão revista em 5/5/2008

Um mapa é, abstratamente, um conjunto de mapeamentos, ou associações, entre um objeto chave e um objeto valor-associado, onde as chaves são únicas. Cada associação chave-valor (*key-value*) implementa a interface Map.Entry.

Um mapa pode ser visualizado como uma tabela com 2 colunas. Cada linha dessa coluna representa uma associação entre um objeto chave, e um objeto valor.

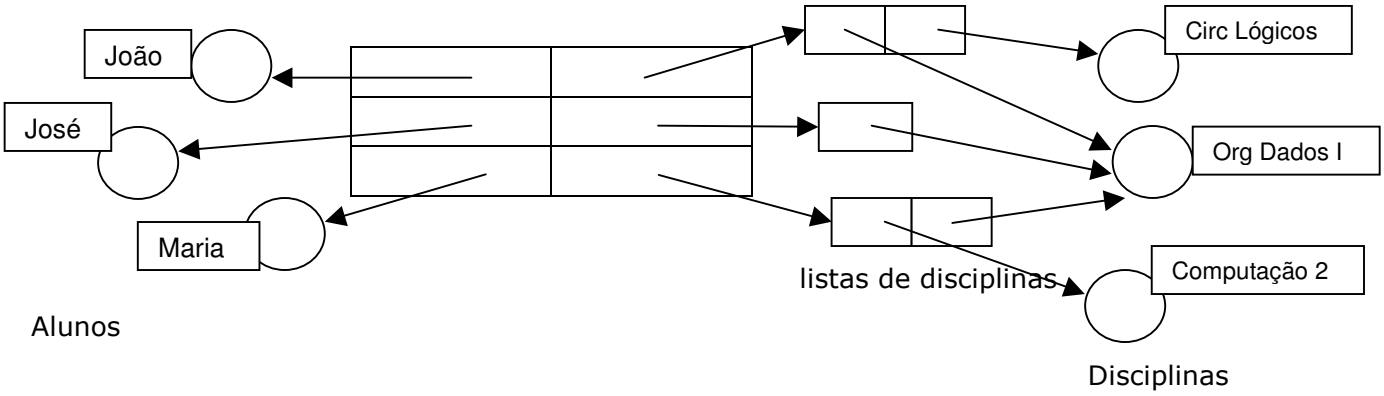
As chaves e os valores devem ser referências a objetos. Valores simples devem ser "empacotados" nas classes correspondentes (int como Integer, etc). Uma associação pode conter uma chave nula, assim como um valor nulo.

Como exemplo de um mapa, suponha uma tabela para associar alunos de um curso às disciplinas que já cursou com aproveitamento. Poderíamos imaginar uma tabela como a abaixo, onde o aluno de nome João cursou duas disciplinas, uma das quais o aluno José curou também:



Esse mapa não é possível, porque a mesma chave (o Aluno João) aparece duas vezes, e as chaves devem ser únicas.

A solução nesse caso é fazer uma associação entre cada aluno e o conjunto de disciplinas que ele cursou:



Agora as chaves são todas distintas, e a figura representa um mapa.

Implementação de Maps em Java

Em Java, esse conceito é implementado como uma interface: java.util.Map

Uma das classes concretas de Java que implementa a interface Map é a classe java.util.HashMap. Outra é java.util.TreeMap.

Extrato básico da API dessa interface:

Notar que alguns parâmetros são do tipo Object, enquanto outros são do tipo da chave ou do tipo do valor.

V	`get(Object key)` Para obter o valor correspondente a uma determinada chave.
V	`put(K key, V value)` Para incluir um par chave-valor
boolean	`containsKey(Object key)` Retorna true se o mapa contém a chave especificada, ou false em caso contrário
boolean	`containsValue(Object value)` Retorna true se o mapa mapeia alguma chave para o valor especificado, ou false em caso contrário
boolean	`isEmpty()` Retorna true se o mapa não contém nenhuma associação.
Set<K>	`keySet()` Retorna uma visão como um Set das chaves contidas nesse mapa (no sentido que não pode ser alterada. Tentativas lançam exceção de UnsupportedOperationException)
V	`remove(Object key)` Remove a entrada desse map para essa chave, caso exista. Retorna o valor correspondente, ou null se a chave não existia no map (operação opcional).
int	`size()` Retorna a quantidade de entradas (pares chave-valor) do map.
Collection<V>	`values()` Retorna uma coleção com os valores contidos no map.

Há ainda:

void	`clear()` Remove todas as entradas do map (operação opcional).
Set<Map.Entry<K,V>>	`entrySet()` Retorna uma *set view* das entradas contidas no map. (um conjunto de Map.Entry que pode ser lido, mas não alterado)
boolean	`equals(Object obj)` Compara o object obj com esse map para checar igualdade.
int	`hashCode()` Retorna o valor do código hash para este map.
void	`putAll(Map<? extends K,? extends V> outroMap)` Copia todas as entradas do map `outroMap` para este map. (operação opcional). Obs: O `outroMap` pode ter chaves que são de subclasse das chaves deste map, e valores que são de subclasse dos valores deste map)

Para criar o mapa do exemplo faríamos:

```
Aluno joao = new Aluno("João");
Aluno jose = new Aluno("José");
Aluno maria = new Aluno("Maria");
Disciplina clog = new Disciplina("Circuitos Lógicos");
Disciplina od1 = new Disciplina("Org. Dados I");
Disciplina comp2 = new Disciplina ("Computação II");
HashMap<Aluno, Disciplina[]> historico = new HashMap<Aluno, Disciplina[]> ();
historico.put(joao,new Disciplina[]{clog,od1});
historico.put(jose, new Disciplina[]{od1});
historico.put(maria, new Disciplina[]{od1,comp2});
```

```
Disciplina[] vetd;
if(historico.containsKey(joao)) {
      vetd = historico.get(joao);
      for(Disciplina d: vetd){
            System.out.println(d);
      }
}
```

Igualdade em Java

Queremos evitar que entradas com chaves duplicadas entrem no Map.

Ou seja, não deve poder existir no Map duas chaves que apontem para objetos iguais.

Mas o que são "dois objetos iguais"?

Em geral estamos interessados em considerar iguais dois objetos que representem a mesma entidade na nossa aplicação, mesmo que não ocupem o mesmo espaço de memória. Por exemplo, duas datas criadas separadamente, mas com os mesmos valores de dia, mês e ano, ou duas strings distintas com exatamente os mesmos caracteres, na mesma ordem.

Em Java, o operador de igualdade == retorna true se e somente se ambos os operandos possuem o mesmo valor. Se forem referências (endereços de memória), o operador == só retorna true se ambas as referências apontarem para o mesmo objeto na memória.

O método equals():
Para poder comparar objetos distintos quanto à igualdade "para fins da aplicação", Java utiliza o método `boolean equals(Object obj)`. Esse método é definido na classe Object com o mesmo sentido que o operador == , e deve ser redefinido em todas as classes onde esse sentido deve ser modificado.

HashMap

Para se poder usar objetos de uma classe K como chaves de um HashMap é preciso observar o seguinte:

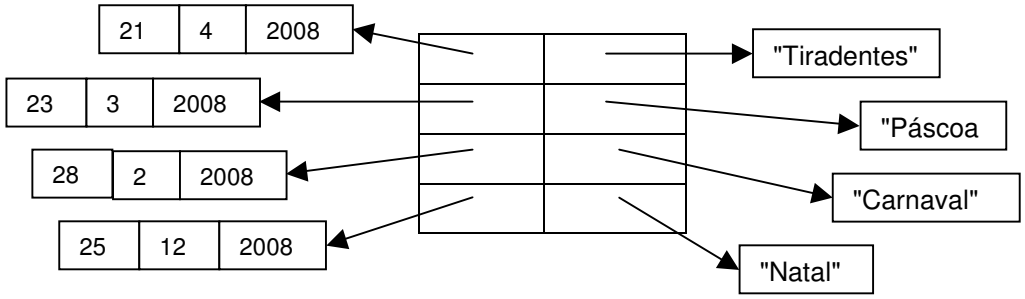
a) Duas chaves iguais a1 e a2 (para fins da aplicação) devem sempre produzir o valor true nas expressões `a1.equals((Object)a2)` e `a2.equals((Object)a1)`, e duas chaves diferentes devem sempre produzir sempre o valor false.

b) o HashMap usa um método `int hashCode()` que é aplicado à chave para calcular um inteiro. Duas chaves iguais a1 e a2 (no sentido acima) devem produzir sempre o mesmo inteiro com `a1.hashCode()` e `a2.hashCode()`. Duas chaves diferentes podem produzir o mesmo inteiro ou não.
   Caso duas chaves iguais (no sentido equals()) produzam inteiros diferentes quando ativarem hashCode() , o HashMap incluirá as duas entradas, violando o principio de não incluir chaves duplicadas.

É necessário então garantir que na classe K da chave existam os métodos int hashCode() e boolean equals(Object obj) redefinidos para terem as características acima.

O exemplo abaixo ajuda a compreender essas exigências:

Suponha uma mapa que associa datas a nomes de feriados.



Suponha que foi criada uma classe Data, cujas instâncias representam datas do calendário:

```
public class Data{
   int  dia, mes, ano;
   public Data(int d, int m, int a){
       dia = d;
       mes = m;
       ano = a;
   }
   etc....
}
```

No caso da tabela de feriados, a chave é uma instância de Data. Quando colocamos a data na tabela, criamos a data com new Date(....). Mas quando formos consultar a tabela, usaremos outra instância de Date, que poderá representar a "mesma" data. Temos situação semelhante à vista acima com as strings.

Portanto, para que objetos da classe Date possam ser usados como chaves, será preciso implementar nesta classe os dois métodos, *equals()* e *hashCode()*, com as propriedades acima descritas.

O método *equals()* é simples. Duas data são iguais se possuem os 3 campos com os mesmos valores:

```
public class Data{
....etc

       public boolean equals(Object obj) {
          Data d = (Data)obj;
          return dia=d.dia && mes==d.mes && ano == d.ano;
       }
}
```

O método *hashCode()* também é simples. Basta que retorne sempre o mesmo inteiro para duas instâncias de Data que representem a mesma data:

```
public class Data{
....etc

       public int hashCode() {
          return dia+mes+ano;
       }
}
```

TreeMap

Um TreeMap<K, V> é uma implementação de Map que garante que as entradas ficarão ordenadas pelas chaves K de acordo com um critério estabelecido. Quando dizemos isso, o significado é que, quando as entradas forem acessadas via um iterador sobre o conjunto das chaves, elas serão visitadas nessa ordem.

O critério de comparação usado pode ser passado para o TreeMap de duas formas:

a) o TreeMap vai usar o critério definido pelo método `int compareTo(K elem)` da classe K das chaves. Nesse caso a classe das chaves K deve implementar a interface Comparable, e o construtor do TreeMap não terá argumento:

```
new TreeMap<K, V> ();
```

O método `int compareTo(K elem)` é chamado de "a ordenação natural dos objetos da classe K". Ou seja, é esse método que vai suprir o critério de ordenação para o algoritmo de ordenação dos elementos da classe. Esse método deve retornar um valor negativo, se *this* deve vir antes de elem, deve retornar um valor zero se *this* deve estar na mesma posição que elem, e deve retornar um valor positivo, se *this* deve estar depois de elem, na ordenação. E também precisa ser consistente com o método equals(), como visto a seguir.

b) o TreeMap vai usar o critério definido pelo método `int compare(K a1, K a2)` de uma classe especialmente construída que deve implementar a interface Comparator<K>. Nesse caso o construtor do TreeMap deve receber como argumento uma instância desse Comparator.
Por exemplo, se a classe do Comparator for:

```
class ComparadorEspecifico implements Comparator<K> {.........}
```

a chamada do construtor deve ser:

```
new TreeMap<K,V> (new ComparadorEspscifico());
```

Consistência entre os métodos equals() e compareTo():

Para utilizar as classes TreeMap e TreeSet é fundamental que o método `compareTo()` seja consistente com o método `equals()`, no sentido de que, para quaisquer dois objetos e1 e e2 da classe <T>, o valor booleano de `e1.equals(e2)` será sempre igual ao valor booleano de `e1.compareTo(e2)==0`.

O motivo é que duas chaves que dão resultado zero com o método `compareTo()` serão consideradas iguais pelo TreeMap, mesmo que o método `equals()` resulte em falso. E se isso ocorrer, a segunda tentativa de inclusão da chave será recusada e poderão ocorrer comportamentos estranhos do TreeMap.

Como exemplo, suponha um TreeMap em que a chave seja do tipo Pessoa, e que nessa classe existam atributos CPF e data de nascimento. Suponha que o método `equals()` considere que duas Pessoas são iguais se tiverem o mesmo CPF.

Suponha que queremos que as entradas do TreeMap fiquem ordenadas pela data de nascimento das pessoas. Nesse caso poderíamos fazer o método `compareTo()` de Pessoa de tal forma a retornar o mesmo que o `compareTo()` da classe Data. O problema aqui é que isso fará com que duas pessoas diferentes com a mesma data de nascimento passem a ser consideradas iguais pelo TreeMap, e não conseguiremos incluir mais de uma entrada onde a pessoa tem a mesma data de nascimento, mesmo com CPF's diferentes. Para resolver esse problema, poderíamos fazer o método `compareTo()` considerar a combinação de data e CPF, algo assim:

```
public boolean equals(Object obj){
   Pessoa p = (Object) obj;
   return cpf.equals(p.getCPF());
}

public int compareTo(Pessoa p){
   if(p.getDataNasc().compareTo(this.getDataNasc())==0){
      return p.getCPF().compareTo(this.getCPF());
   else return p.getDataNasc().compareTo(this.getDataNasc());
 }
```

Note que esse método é consistente com `equals()`, pois só retornará 0 se as duas pessoas forem iguais, no sentido de terem o mesmo CPF. E também garante a ordenação das pessoas por ordem crescente de data de nascimento.

Caso o TreeMap use um Comparator no seu construtor, o método `compare()` do Comparator também deve ser consistente com `equals()`, ou o mesmo problema poderá ocorrer.

OBS: todos os cuidados aqui descritos para usar Maps valem para os Sets correspondentes (HashSet e TreeSet).