

L'interface **Map** *et* *son implémentation* *par une* **HashMap**

Université de Montpellier
Faculté des sciences
Mars 2021

L'interface **Map**

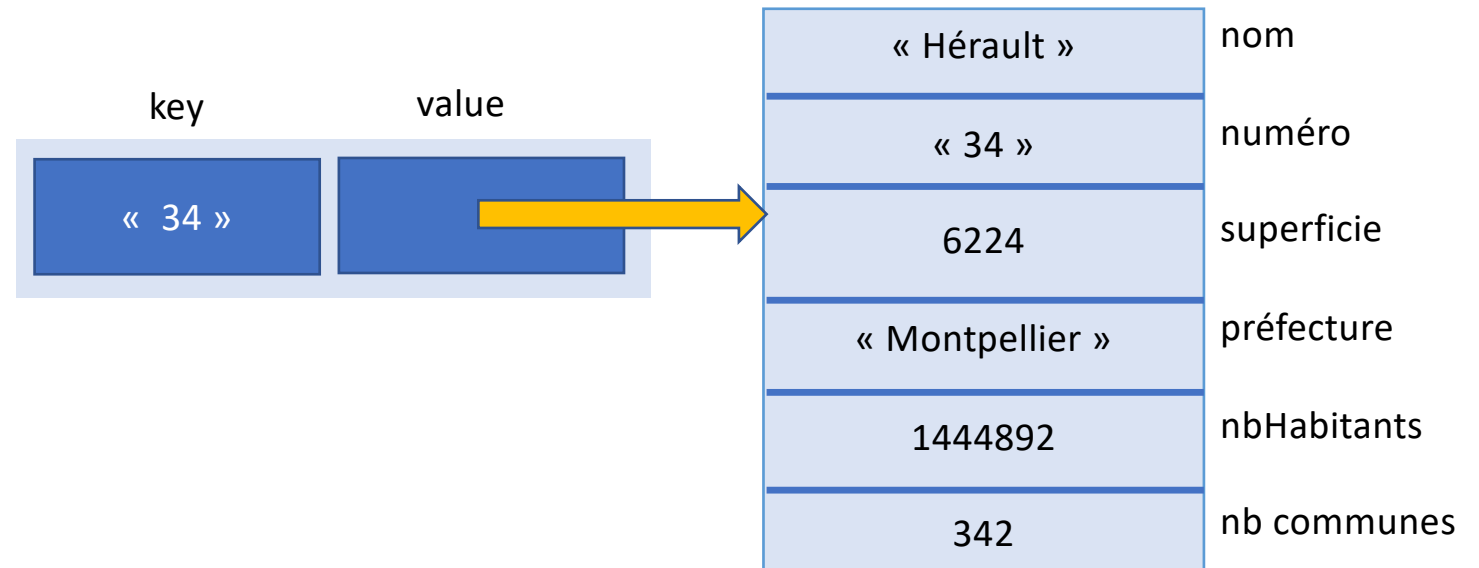
- **Map** = Dictionnaire **associatif**
- Type abstrait de données
- Modéliser des fonctions (mathématique) partielles discrètes d'un ensemble K (clefs) dans un ensemble V (valeurs)
 - Cela **associe** à un élément de K une valeur de V
- Opérations principales
 - **put (clef, valeur)** Insérer une **association** (clef, valeur)
 - **get(clef)** Rechercher par la clef (pour obtenir la valeur)
 - **remove(clef)** Supprimer une association (clef, valeur)

Exemple de dictionnaire associatif

- Associer la description des départements français à leur numéro
- 101 départements
 - Numéro à 2 ou 3 chiffres
 - Pouvant commencer par 0
 - Pouvant contenir une lettre (ex. 2A, 2B pour les départements corses)
 - Tous les numéros n'existent pas
 - N'existent pas : 20, 985, rien entre 95 et 971, rien entre 978 et 984, rien après 989
- Clef : numéro de département (String)
- Valeur : description du département (Département)

Associer la description des départements français à leur numéro

- Clef : numéro de département (String)
- Valeur : description du département (Département)



Deux implémentations des Maps (structures de données)

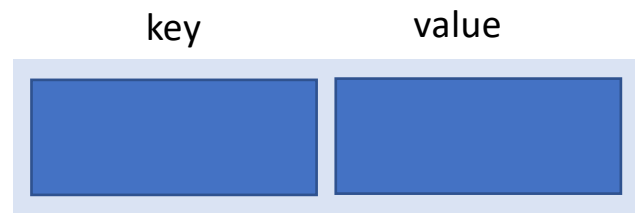
- Par une table de hachage (**HashMap**) : c'est l'objet de ce cours
- Par un arbre (**TreeMap**) : nous verrons les arbres à un prochain cours

Principe simplifié de la HashMap

- Des Associations (appelées aussi Paire, ou **Entrée / Entry**) sont créées pour stocker les couples (clef, valeur)
- Elles sont placées dans un **tableau**
- La clef permet de trouver « rapidement » la case du tableau où est rangée la valeur

Principe simplifié de la HashMap

- Des Associations (appelées aussi Paires, ou **Entrées**) sont créées pour stocker les couples (clef, valeur)

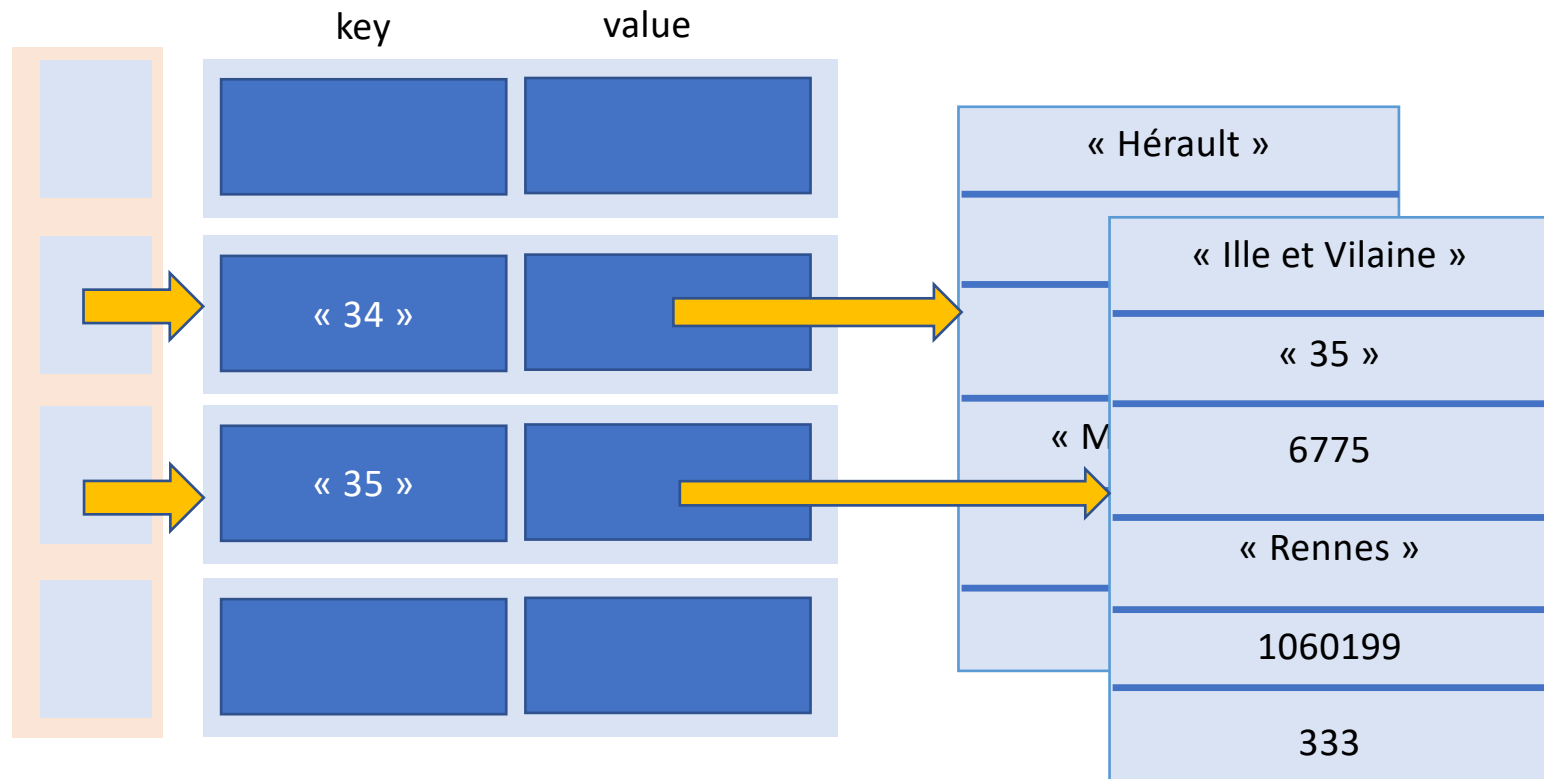


Représentation des entrées (associations)

```
public class MyEntry<K,V> {  
    K key;  
    V value;  
  
    public MyEntry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
}
```


Principe simplifié de la HashMap

- Les associations sont placées dans un **tableau**



Principe simplifié de la HashMap

```
public class MyHashMap<K,V>
    implements Map<K,V>{
    // pour stocker les associations
    private MyEntry<K,V>[] table;
    private int size;

    public MyHashMap() {
        this.table = new MyEntry[10];
    }

    public V put(K key, V value){...}
    public V get(Object key)
    // ...
}
```

Principe simplifié de la HashMap

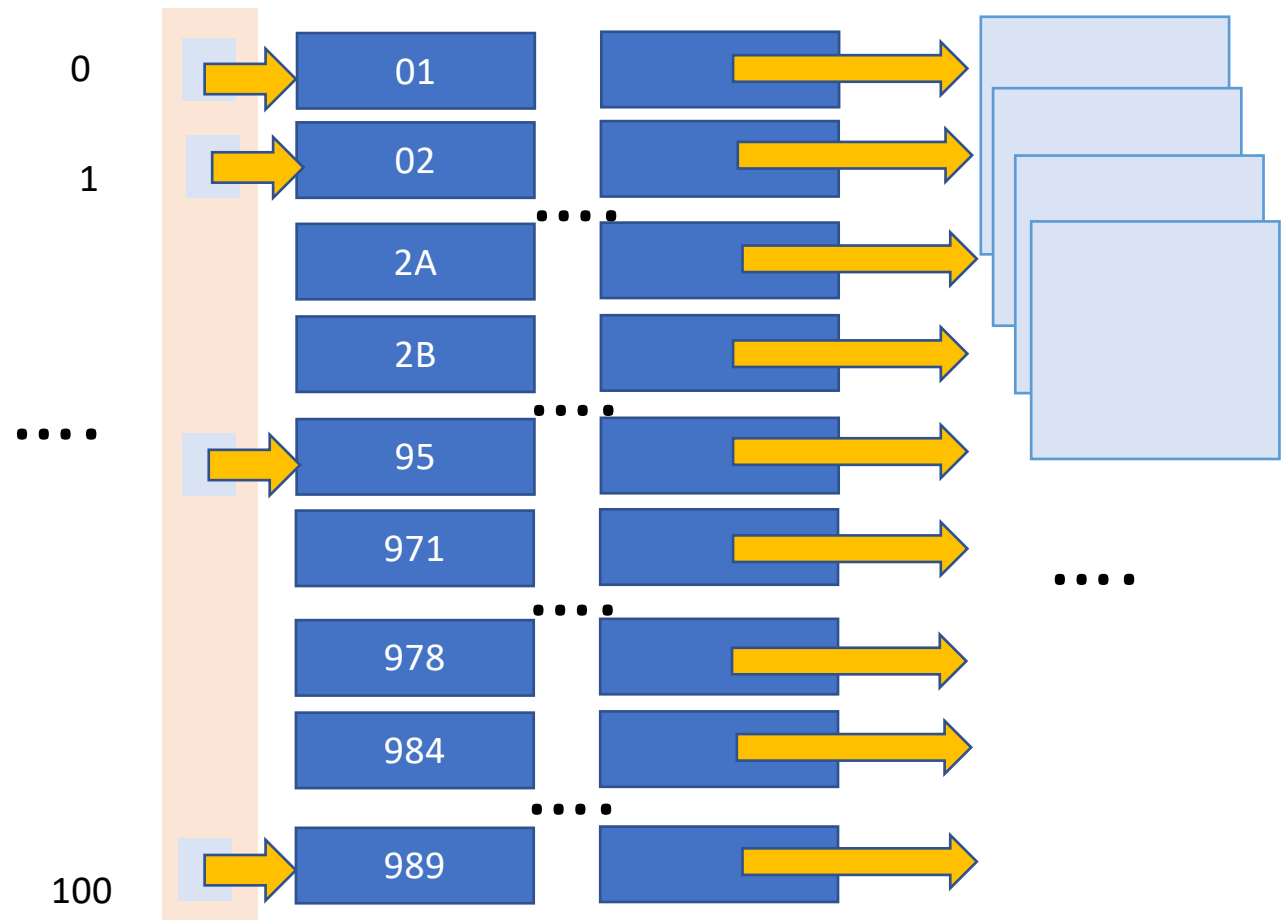
- La clef permet de trouver (**get**) « rapidement » la case du tableau où est rangée la valeur
- Pour cela il faut une « **formule magique** », pour passer de la clef à l'endroit (indice dans la **table**) où on va ranger (**put**) ou rechercher (**get**) l'association correspondante
- Idéalement

table[formuleMagique(key)] contient la valeur

table[formuleMagique(« 34 »)] contient la description de l'hérault (l'objet représentant l'hérault)

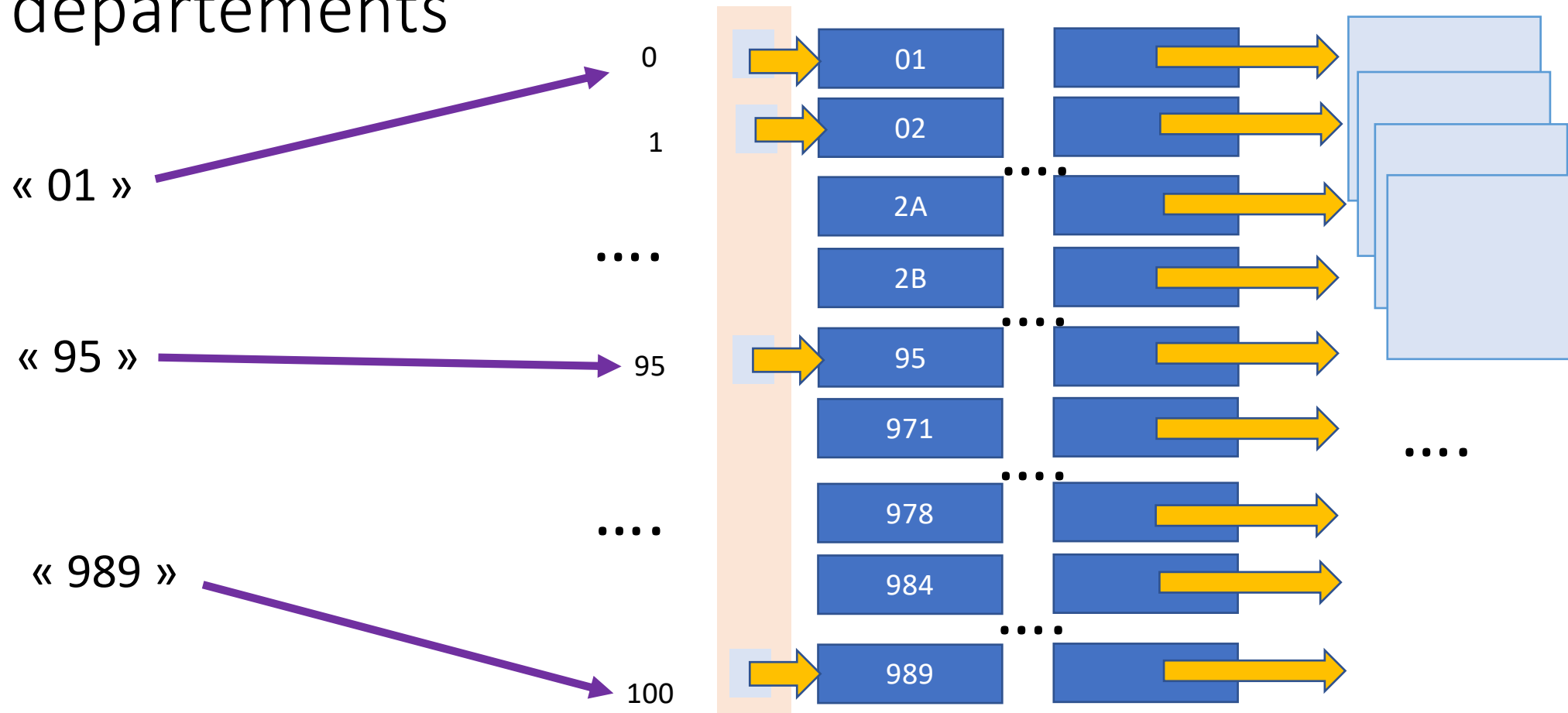
Formule magique pour la HashMap de départements

- 101 départements
- **Indices** de 0 à 100
- Passer des **clefs** = numéros de département (String) aux indices



- 01 : Ain
- 02 : Aisne
- 03 : Allier
- 04 : Alpes-de-Haute-Provence
- 05 : Hautes-Alpes
- 06 : Alpes-Maritimes
- 07 : Ardèche
- 08 : Ardennes
- 09 : Ariège
- 10 : Aube
- 11 : Aude
- 12 : Aveyron
- 13 : Bouches-du-Rhône
- 14 : Calvados
- 15 : Cantal
- 16 : Charente
- 17 : Charente-Maritime
- 18 : Cher
- 19 : Corrèze
- 2A : Corse-du-Sud
- 2B : Haute-Corse
- 21 : Côte-d'Or
- 22 : Côtes-d'Armor
- 23 : Creuse
- 24 : Dordogne
- 25 : Doubs
- 26 : Drôme
- 27 : Eure
- 28 : Eure-et-Loir
- 29 : Finistère
- 30 : Gard
- 31 : Haute-Garonne
- 32 : Gers
- 33 : Gironde
- 34 : Hérault
- 35 : Ille-et-Vilaine
- 36 : Indre
- 37 : Indre-et-Loire
- 38 : Isère
- 39 : Jura
- 40 : Landes
- 41 : Loir-et-Cher
- 42 : Loire
- 43 : Haute-Loire
- 44 : Loire-Atlantique
- 45 : Loiret
- 46 : Lot
- 47 : Lot-et-Garonne
- 48 : Lozère
- 49 : Maine-et-Loire
- 50 : Manche
- 51 : Marne
- 52 : Haute-Marne
- 53 : Mayenne
- 54 : Meurthe-et-Moselle
- 55 : Meuse
- 56 : Morbihan
- 57 : Moselle
- 58 : Nièvre
- 59 : Nord
- 60 : Oise
- 61 : Orne
- 62 : Pas-de-Calais
- 63 : Puy-de-Dôme
- 64 : Pyrénées-Atlantiques
- 65 : Hautes-Pyrénées
- 66 : Pyrénées-Orientales
- 67 : Bas-Rhin
- 68 : Haut-Rhin
- 69D : Rhône
- 69M : Métropole de Lyon
- 70 : Haute-Saône
- 71 : Saône-et-Loire
- 72 : Sarthe
- 73 : Savoie
- 74 : Haute-Savoie
- 75 : Paris
- 76 : Seine-Maritime
- 77 : Seine-et-Marne
- 78 : Yvelines
- 79 : Deux-Sèvres
- 80 : Somme
- 81 : Tarn
- 82 : Tarn-et-Garonne
- 83 : Var
- 84 : Vaucluse
- 85 : Vendée
- 86 : Vienne
- 87 : Haute-Vienne
- 88 : Vosges
- 89 : Yonne
- 90 : Territoire de Belfort
- 91 : Essonne
- 92 : Hauts-de-Seine
- 93 : Seine-Saint-Denis
- 94 : Val-de-Marne
- 95 : Val-d'Oise
- 971 : Guadeloupe
- 972 : Martinique
- 973 : Guyane
- 974 : La Réunion
- 975 : Saint-Pierre-et-Miquelon
- 976 : Mayotte
- 977 : Saint-Barthélemy
- 978 : Saint-Martin
- 984 : Terres australes et antarctiques françaises
- 986 : Wallis-et-Futuna
- 987 : Polynésie française
- 988 : Nouvelle-Calédonie
- 989 : Île de Clipperton

Formule magique pour la HashMap de départements



Formule magique pour la HashMap de départements

- Pour les clefs **c** de « 01 » à « 19 »
 Integer.valueOf(« 01 ») = 1 String -> int
 Integer.valueOf(c)-1 donne l'entier correspondant -1
 Integer.valueOf(« 34 ») donne 33
- Pour c = « 2A » ce sera l'indice : 19
- Pour c = « 2B » ce sera l'indice : 20
- Pour les clefs **c** de « 21 » à « 68 »
 Integer.valueOf(c)
- Etc.

Ici c'est facile car le tableau est de la bonne taille et on sait comment ranger les départements par ordre croissant de numéro, le hachage est parfait !

Mais comment faire en général ?

- Chaque type de clefs pourrait avoir sa propre formule magique
 - On l'appellera la **fonction de hachage**
 - **hashCode()** en Java
- Si on ne connaît pas à l'avance le nombre d'associations à stocker, il sera difficile d'avoir un tableau qui soit exactement de la bonne taille
- Il y a une relation entre la fonction de hachage et la taille du tableau puisque la fonction de hachage doit retourner un indice dans le tableau à partir de la clef
- On recherchera une fonction de hachage injective : telle que deux clefs différentes ont un résultat de hachage différent, mais ce sera difficile et rarement le cas
- situation générale : la fonction n'est pas injective, on aura des **collisions**

Fonction injective (ex. wikipedia)

- Prenons le cas d'une station de vacances où un groupe de touristes doit être logé dans un hôtel. Chaque façon de répartir ces touristes dans les chambres de l'hôtel peut être représentée par une application de l'ensemble des touristes, X , vers l'ensemble des chambres, Y (à chaque touriste est associée une chambre).
- Les touristes souhaitent que l'application soit **injective**, c'est-à-dire que *chacun d'entre eux ait une chambre individuelle*. Cela n'est possible que si le nombre de touristes ne dépasse pas le nombre de chambres.
- C'est un cas idéal mais pour les HashMap ce cas idéal ne se produira pas toujours
- fonction hash : clefs vers les entiers (indices des cases du tableau)

- En Java la classe Object fournit une fonction de hachage « par défaut » `public int hashCode()`
- Cette fonction transforme l'adresse interne de l'objet en un entier (destiné à être un indice dans la `table`)
- **Elle ne sera pas parfaite (non injective), certains objets auront la même valeur de hashCode()**
- Elle peut être redéfinie dans chaque classe de manière à être plus efficace, par exemple dans les cas où on peut avoir un hachage parfait (ex. département)
- Elle est redéfinie par exemple dans la classe `String` par un calcul utilisant les codes des caractères et la longueur de la chaîne

Fonction de hachage

- Idée de base pour faire put(key, value)

`table[key.hashCode()%table.length] = value`

- Idée de base pour faire get(key)

`return table[key.hashCode ()%table.length]`

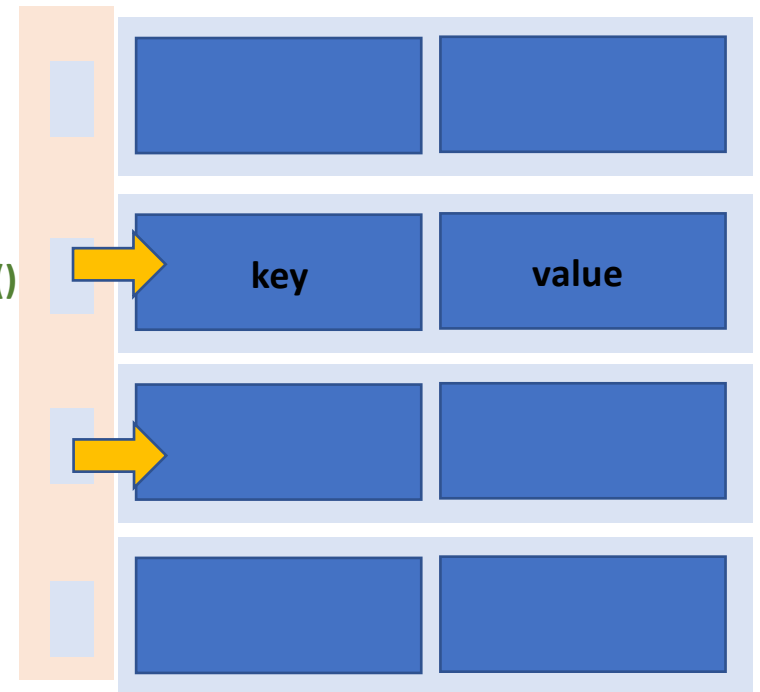
Indices

0

`key.hashCode()
%table.length`

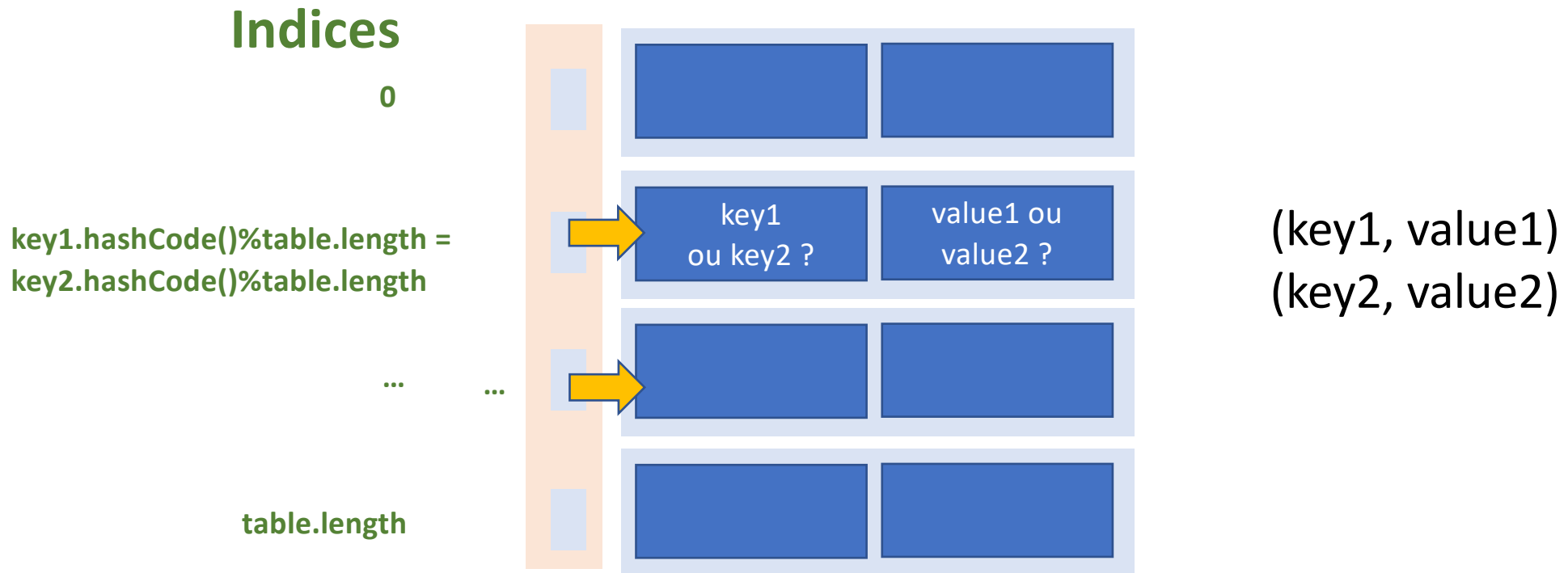
...

`table.length`



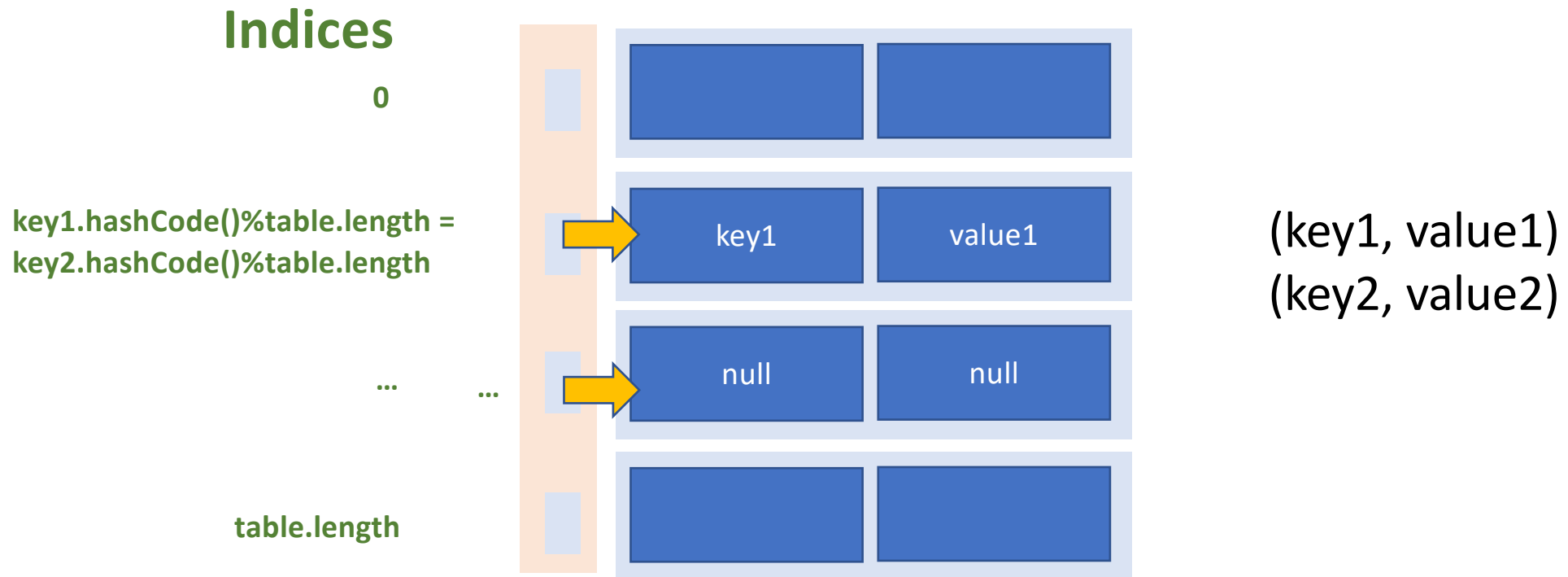
Fonction de hachage

- Mais si la fonction de hachage n'est pas parfaite, on a des **collisions**, quand 2 clefs ont la même valeur de $\text{hashCode} \% \text{table.length}$



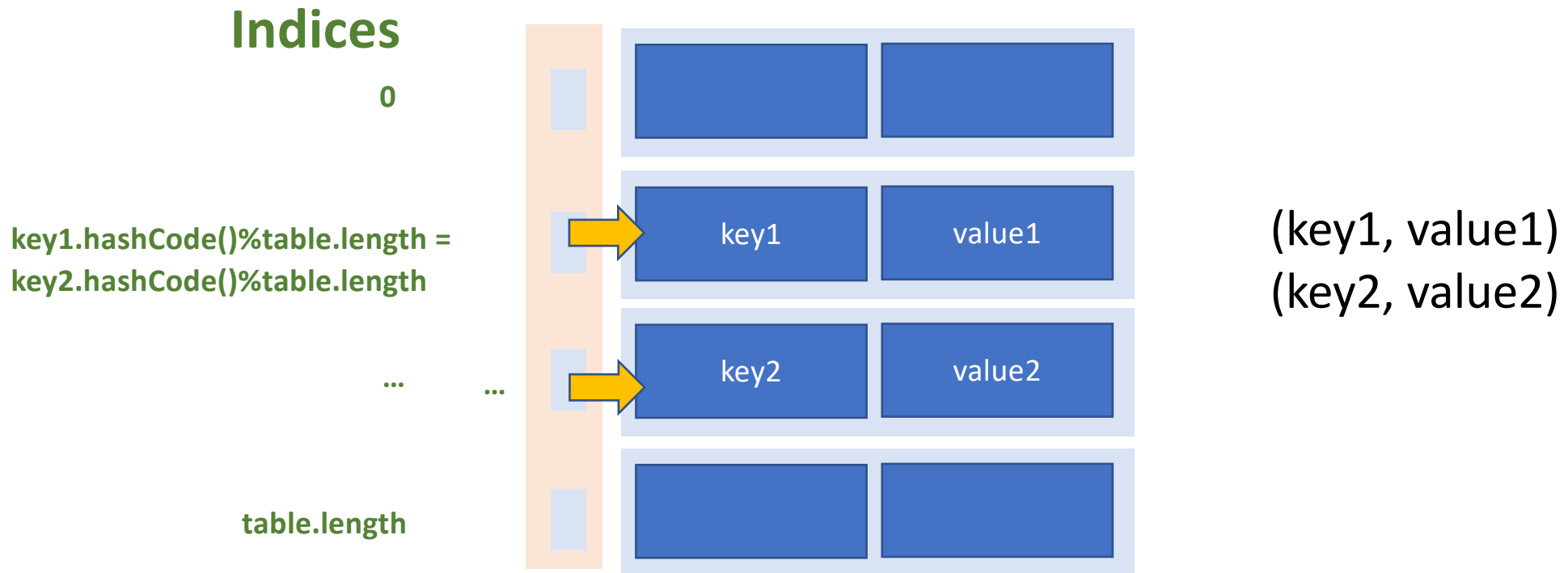
Fonction de hachage

- La première entrée (key1, value1) est rangée à l'endroit prévu donné par $\text{key1.hashCode() \% table.length}$



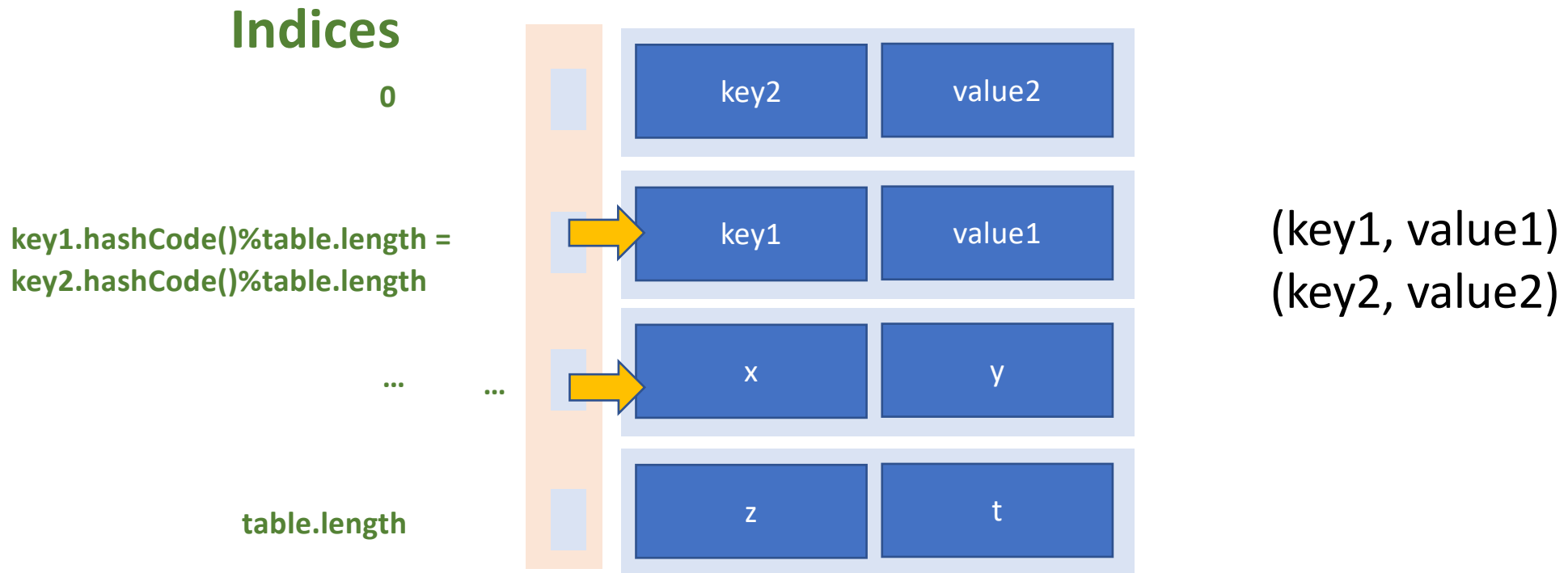
Fonction de hachage

- Pour ranger la seconde entrée (key2, value2), une stratégie simple consiste à **chercher la première case libre juste après l'indice $\text{key2.hashCode() \% table.length}$** , ici elle est juste après mais parfois il faut avancer plus loin



Fonction de hachage

- Pour ranger la seconde entrée (key2, value2), une stratégie simple consiste à chercher la première case libre juste après l'indice $\text{key2.hashCode() \% table.length}$, ici elle est juste après mais parfois il faut avancer plus loin, **et parfois même revenir au début du tableau**



```
public class MyHashMap<K,V> implements Map<K,V>{

    private MyEntry<K,V>[] table; private int size;

    public V put(K key, V value) {
        if (table.length == this.size()) // on agrandit parfois qd 75% est
                                           //occupé ici c'est si 100%

        // tableau plein
            this.agrandir();
        int hashCourant = key.hashCode()%table.length;
        while (table[hashCourant]!=null) {
            hashCourant++;
            if (hashCourant == table.length)
                //hashCourant = hashCourant%table.length; ou ...
                hashCourant = 0;
        }// ici hashcourant est un indice vers une case vide
        table[hashCourant]= new MyEntry(key, value);
        size++;
        return value;
    }
}
```


Création de départements

```
Departement H = new Departement("Hérault", "34", 6224, "Montpellier", 1444892, 342);  
Departement I = new Departement("Ille et Vilaine", "35", 6775, "Rennes", 1060199, 333);  
Departement HL = new Departement("Haute Loire", "43", 4977, "Le Puy en Velay", 227283, 257);  
Departement CS = new Departement("Corse-du-Sud", "2A", 4014, "Sartène", 157249, 124);  
Departement HC = new Departement("Haute-Corse", "2B", 4666, "Bastia", 177689, 236);  
Departement HG = new Departement("Haute-Garonne", "31", 6309, "Toulouse", 1362672, 587);  
Departement G = new Departement("Gers", "32", 6257, "Auch", 191091, 462);  
Departement Gi = new Departement("Gironde", "33", 9975, "Bordeaux", 1583384, 535);  
Departement Ga = new Departement("Gard", "30", 5853, "Nîmes", 744178, 351);  
Departement Au = new Departement("Aube", "10", 6004, "Troyes", 310020, 431);  
Departement Ma = new Departement("Marne", "51", 8169, "Châlons-en-Champagne", 568895, 616);
```

Création d'une HashMap

```
MyHashMap<String,Departement> mhm =  
    new MyHashMap<>( );
```

```
// Ajout successif des entrées (numéro de département, département)
```

```
mhm.put("34", H);
```

```
mhm.put("35", I);
```


```
// ...
```

Le dictionnaire vide

*Nota : pour simplifier la représentation des objets
la table est figurée de cette manière*

0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

put(« 34 »,H)

« 34 ».hashCode()%10=3 

```
System.out.println("34".hashCode());
```

1633

```
System.out.println(1633%10);
```

3

*Nota : pour simplifier la représentation des objets
département est seulement figurée .*

*La table contient en réalité l'adresse de l'objet
Comme montré dans la diapo suivante*

0	null	
1	null	
2	null	
3	34	[nom=Hérault, numero=34,...]
4	null	
5	null	
6	null	
7	null	
8	null	
9	null	

put(« 34 »,H)

« 34 ».hashCode()%10=3

Nota :
La table contient
en l'adresse de
l'entrée comme
montré ici

0	null
1	null
2	null
3	
4	null
5	null
6	null
7	null
8	null
9	null

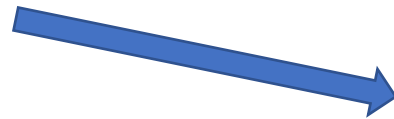


H

« Hérault »
« 34 »
6224
« Montpellier »
1444892
342

put(« 35 »,l)

« 35 ».hashCode()%10=4



*Nota : pour simplifier la représentation des objets
département est seulement figurée par le début de
la structure.*

La table contient en réalité l'adresse de l'entrée

0	null	
1	null	
2	null	
3	34	[nom=Hérault, num=34,...]
4	35	[nom=Ille et Vilaine, num=35]
5	null	
6	null	
7	null	
8	null	
9	null	

put(« 43 »,HL)

« 43 ».hashCode()%10=3
cas de non injectivité

La case 3 est déjà occupée !

COLLISION

*Stratégie : on avance jusqu'à la
première case libre*

0	null	
1	null	
2	null	
3	34	[nom=Hérault, num=34,...]
4	35	[nom=Ille et Vilaine, num=35,..]
5	null	
6	null	
7	null	
8	null	
9	null	

put(« 43 »,HL)

« 43 ».hashCode()%10=3

La première case libre est la 5

On y place (« 43 », HL)

0	null	
1	null	
2	null	
3	34	[nom=Hérault, num=34,...]
4	35	[nom=Ille et Vilaine, num=35,..]
5	43	[nom=Haute Loire, num=43,...]
6	null	
7	null	
8	null	
9	null	

On cherche « 31 »... (get)

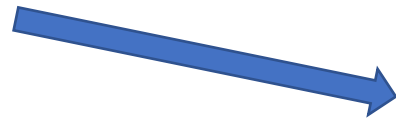


« 31 ».hashCode()%10=3

0	31	[nom=Haute-Garonne, num=31, ...]
1	32	[nom=Gers, num=32, ...]
2	33	[nom=Gironde, num=33, ...]
3	34	[nom=Hérault, num=34,...]
4	35	[nom=Ille et Vilaine, num=35,..]
5	43	[nom=Haute Loire, num=43,...]
6	2A	[nom=Corse-du-sud, num=2A,...]
7	2B	[nom=Haute-Corse, num=2B,...]
8	10	[nom=Aube, num=10,...]
9	30	[nom=Gard, num=30,...]

put(« 51 »,M)

« 51 ».hashCode()%10=2



Mais il n'y a plus de place !

On crée un tableau plus grand

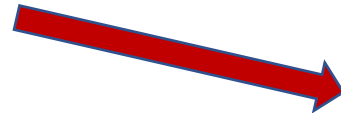
On y replace les entrées

0	31	[nom=Haute-Garonne, num=31, ...
1	32	[nom=Gers, num=32, ...
2	33	[nom=Gironde, num=33, ...
3	34	[nom=Hérault, num=34,...]
4	35	[nom=Ille et Vilaine, num=35,.]
5	43	[nom=Haute Loire, num=43,...]
6	2A	[nom=Corse-du-sud, num=2A,...]
7	2B	[nom=Haute-Corse, num=2B,...]
8	10	[nom=Aube, num=10,...]
9	30	[nom=Gard, num=30,...]

Un tableau plus grand (20 cases)

Pour remplacer les entrées on passe par $\text{hashcode} \% 20$

« 34 ».hashCode() $\%20 = 13$



Alors que l'on avait

« 34 ».hashCode() $\%10 = 3$

On pourra ajouter la nouvelle association !

0	null	null
1	null	null
2	null	null
3	null	null
4	null	null
5	null	null
...
13	34	[nom=Hérault, num=34,...]
...
17	null	null
18	null	null
19	null	null

Autres opérations

- **public V get(Object key)**
 - Retournera la valeur associée à la clef **key** (opération inverse de **put**)
- **public boolean containsKey(Object key)**
 - Retourne vrai si **key** apparaît dans le dictionnaire
- **public boolean containsValue(Object value)**
 - Retourne vrai si **value** apparaît dans le dictionnaire
- **public Set<K> keySet()**
 - Retourne l'ensemble de clefs apparaissant dans le dictionnaire
- **public Collection<V> values()**
 - Retourne l'ensemble des valeurs apparaissant dans le dictionnaire
- **size, isEmpty**
- Et d'autres méthodes comme **remove**

Synthèse

- Un dictionnaire (avec table de hachage) est fait pour être efficace (en temps de calcul) dans :
 - L'ajout
 - Le retrait
- Il n'est pas efficace pour des parcours séquentiels
- Pour l'implémentation HashMap, la place occupée est uniquement l'ensemble des valeurs et des clefs
- Et pour aller plus loin, une implémentation open source en Java 1.7
<http://www.docjar.com/html/api/java/util/HashMap.java.html>
- Travaux à réaliser :
 - Débutants et avancés : classe promotion avec une Map (voir Moodle)
 - Avancés : compléter [//https://repl.it/join/lyylkptp-mariannehuchard](https://repl.it/join/lyylkptp-mariannehuchard)

(Nota : variante d'implémentation dans le TD pour les avancés : 2 tableaux)