# Tables de hachage

CHAPITRE 5(WEISS)

CHAPITRE BIG JAVA, C. HORSTMANN

NOTES DE COURS, É. BAUDRY

# Motivation

| Structure | Accès direct | Insertion | Recherche |
|---|---|---|---|
| Tableau | O(1) | O(n) | O(n) |
| Liste | - | O(1) | O(n) |
| Arbre de recherche équilibré | - | O(log n) | O(log n) |

# Motivation

- Comment faire mieux ?

| Structure | Accès direct | Insertion | Recherche |
|---|---|---|---|
| Tableau | O(1) | O(n) | O(n) |
| Liste | - | O(1) | O(n) |
| Arbre de recherche équilibré | - | O(log n) | O(log n) |
| Table de hachage | - | O(1) | O(1) |

# Motivation

```
public class Main {
    public static String description(String note){
        if(note.equals("A")) return "Excellent";
        if(note.equals("B")) return "Très bien";
        if(note.equals("C")) return "Bien";
        if(note.equals("D")) return "Passable";
        if(note.equals("E")) return "Echèc";
        if(note.equals("X")) return "Abandon";
        return "Erreur";
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.println(description(s.next()));
    }
}
```

# Motivation

```
public class Main {
        public static void main(String[] args) {
         //TreeMap - arbre rouge-noir
         TreeMap<String,String> table = new TreeMap<>();
         table.put("A","Excellent");
         table.put("B","Très bien");
         table.put("C","Bien");
         table.put("D","Passable");
         table.put("E","Echèc");
         table.put("X","Abandon");
         Scanner s = new Scanner(System.in);
         System.out.println(table.get(s.nextLine()));
    }
}
```

# Motivation

```
public class Main {
    public static void main(String[] args) {
        String [] table = new String[256];
        table['A']="Excellent";
        table['B']="Très bien";
        table['C']="Bien";
        table['D']="Passable";
        table['E']="Echèc";
        table['X']="Abandon";
        Scanner s = new Scanner(System.in);
        String inp = s.next();
        System.out.println(table[inp.charAt(0)]);
    }
}
```

# Motivation

```
public class Main {
    public static void main(String[] args) {
        String [] table = new String[26];
        table['A'-'A']="Excellent";
        table['B'-'A']="Très bien";
        table['C'-'A']="Bien";
        table['D'-'A']="Passable";
        table['E'-'A']="Echèc";
        table['X'-'A']="Abandon";
        Scanner s = new Scanner(System.in);
        String inp = s.next();
        System.out.println(table[inp.charAt(0)-'A']);
    }
}
```

# Motivation

- Associer à un sigle une chaîne de caractères
- Sigle
  - IFT2015 – Structures de données
  - 3 lettres 4 chiffres
- Taille: $26^3 * 10^4 = 175\,760\,000 \approx 175M$
- Mémoire: 175M*taille de référence (adresse) = 175M*8 ≈1.31Go
- Code permanant
  - 4 lettres+6 chiffres
- Taille: $26^4 * 10^6 = 456\,976\,000\,000 \approx 425G$
- Besoin en mémoire: 1731Go

# Motivation

- Un objet est une valeur
- La taille du domaine des valeurs possibles (nombre d'objets différents) peut être très grand
- Nombre de chaînes de longueur 8 avec [A-Z]
  - $26^8 = 208827064576 \approx$2.09E11
- Nombre de chaînes de longueur 8 avec [a-zA-Z0-9]
  - $(2*26 + 10)^8 \approx$2.18E14
- Nombre de chaînes de caractères de longueur 10
  - $256^{10} \approx$1.2E24

# Motivation

- Observation
  - Généralement, le nombre d'objets (n) qu'on veut réellement stocké est de plusieurs ordre de grandeurs inférieur à la taille du domaine (D)
- Exemple avec les codes permanents
  - Nombre d'étudiants dans une université: 40 000
  - Nombre d'étudiants dans un cours: 60

# Réduction de la taille d’adressage

- Calculer une adresse dans une table
- Fraction de la clé est utilisée
- Fonctions
  - Division:  /
  - Modulo: %

# Adressage réduit

- La taille d'une table de hachage peut avoir le même ordre de grandeur que l'ensemble de valeurs à stocker
- Pour savoir à quelle adresse dans le tableau une clé est associée, une fonction de réduction est utilisée

# Objectif

- Éviter que des objets ayant des valeurs proches obtiennent la même adresse réduite
- Idée: construire une fonction le plus chaotique possible
- Fonction chaotique: une faible variation de l’entrée entraîne une grande variation en sortie

# Collision

- Se produit quand deux objets ayant des valeurs différentes génèrent la même adresse réduite
- Inévitable en pratique
- Et ce, même si une bonne fonction de hachage
- Besoin d’un mécanisme de gestion de collisions

# Collision

- Stratégies

- Ouverte
  - Linéaire
  - Quadratique

- Structure externe
  - Liste chaînée
  - Arbre binaire de recherche

# Collision

- Adressage Ouvert consiste à résoudre une collision en utilisant une autre entrée (adresse alternative)
- Adresse alternative peut être calculée de façon
  - Linéaire
  - Quadratique
  - À l'aide d'une autre fonction de hachage

- Exemples au tableau

# Tables de hachage

- **Hachage** peut être utilisé pour retrouver un élément dans une structure de données rapidement sans faire la recherche linéaire
- Une table de hachage peut être utilisée pour implémenter des ensembles(set) et des cartes (map)
- La **fonction de hachage** calcule un nombre entier, appelé code de hachage, pour chacun des éléments
- La bonne fonction d'hachage minimise les collisions — les codes de hachage identiques pour les objets différents
- Pour calculer le code de hachage de l'objet `x`:

```
int h = x.hashCode();
```

# Codes de hachage résultant de la fonction `hachCode`

| String | Hash Code |
|---|---|
| "Adam" | 2035631 |
| "Eve" | 700068 |
| "Harry" | 69496448 |
| "Jim" | 74478 |
| "Joe" | 74656 |
| "Juliet" | -2065036585 |
| "Katherine" | 2079199209 |
| "Sue" | 83491 |

# Implémentation simpliste d'une table de hachage

- Pour implémenter
  - *Générer les codes de hachage des objets*
  - *Créer un tableau*
  - *Insérer chaque objet dans une position correspondante à son code de hachage*
- Pour tester si l'objet est présent dans un ensemble
  - *Calculer son code de hachage*
  - *Vérifier si la position correspondante à ce code de hachage est déjà occupée*

# Implémentation simpliste d'une table de hachage

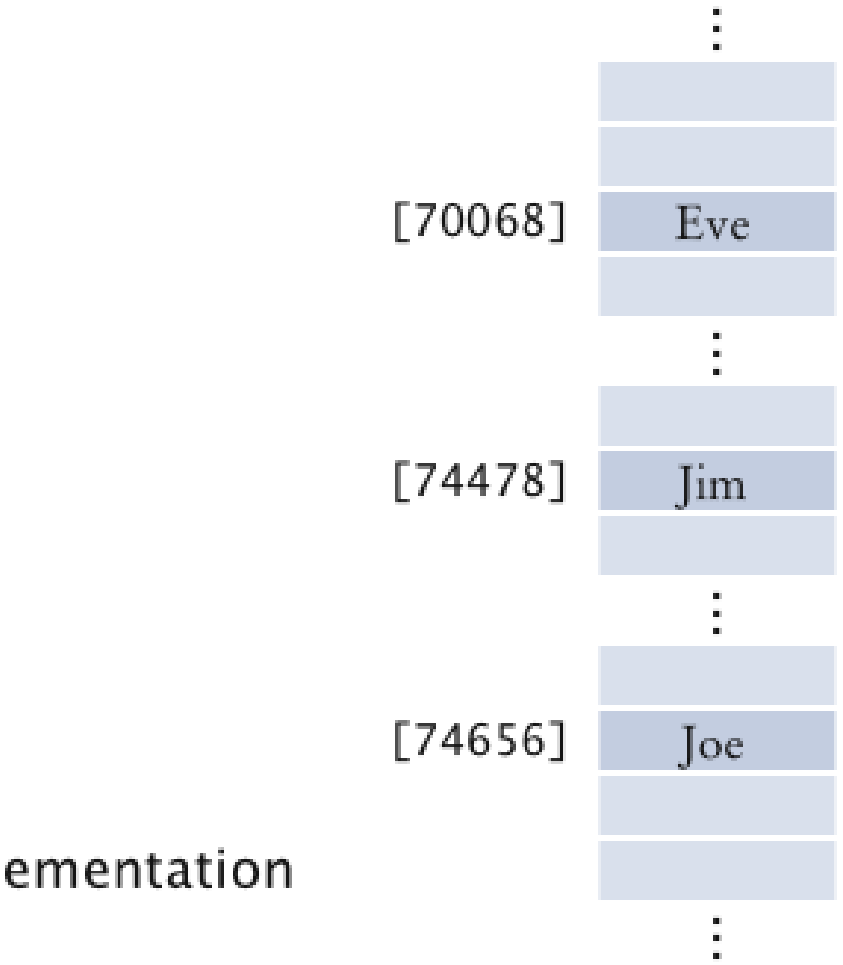



**Figure 5**
A Simplistic Implementation of a Hash Table

# Problèmes avec l'implémentation simpliste

- Il n'est pas possible d'allouer un tableau assez grand pour contenir tous les indexes entiers possibles
- Il est possible que deux objets différents pourront avoir le même code de hachage

# Solutions

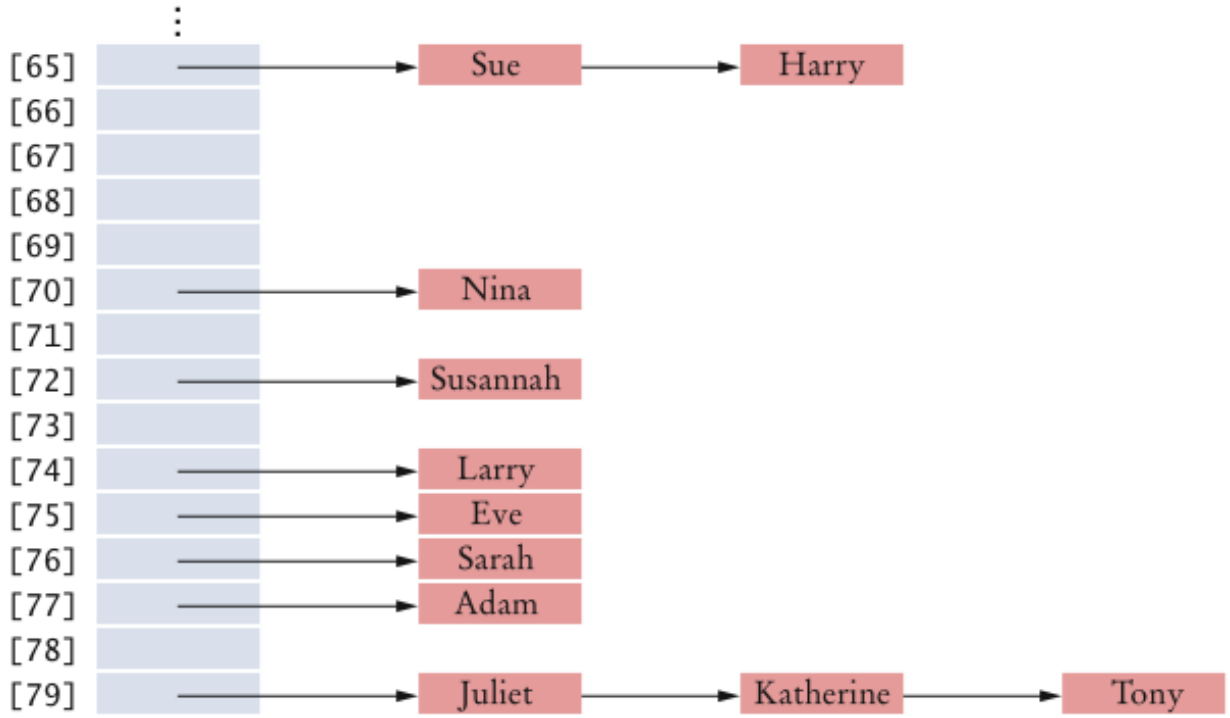
- Choisissez une taille raisonnable du tableau, réduisez le nombre de codes de hachage selon la taille du tableau

```
int h = x.hashCode();
if (h < 0) h = -h;
position = h % buckets.length;
```

- Lorsque les éléments ont le même code:
  - *Utiliser une séquence des nœuds pour stocker des nombreuses objets dans une même position du tableau*
  - *Ces séquences des nœuds sont appelées des paniers (buckets)*

# Une table de hachage avec des paniers

**Figure 6**
A Hash Table with Buckets to Store Elements with the Same Hash Code



# Algorithme pour retrouver l'objet `x` dans une table de hachage

1. Avoir un index `h` dans la table de hachage
    - *Calculer le code de hachage*
    - *Réduire ce code par un modulo du nombre total de paniers*
2. Itérer à travers des éléments du panier dans la position `h`
    - *Pour chaque élément du bucket, comparer s'il est égale à `x`*
3. Si on trouve l'égalité, donc `x` est dans l'ensemble
    - *Sinon, `x` n'est pas dans l'ensemble*

# Tables de hachage

- Une table de hachage peut être implémentée comme un tableau des paniers
- Les paniers sont des séquences de nœuds contenants les éléments avec le même code de hachage
- S'il y a peu de collisions, l'addition suppression et recherche d'un élément prend le temps constant
  - *O(1)*
- Pour que cet algorithme soit efficace, la taille des paniers doit être petite
- La taille de la table doit être un nombre premier plus grand que le nombre d'éléments prévu
  - *Un excès de la capacité de 30% est typiquement recommandé*

# Tables de hachage

- Ajouter un élément: extension simple de l'algorithme de recherche de l'objet
  - *Calculer le code de hachage pour localiser le panier où l'élément doit être inséré*
  - *Essayez de retrouver l'élément dans ce panier*
  - *Si l'élément est présent, faites rien, sinon, insérez-le*
- Supprimer un élément
  - *Calculer le code de hachage pour localiser le panier où l'élément doit être inséré*
  - *Essayez de retrouver l'élément dans ce panier*
  - *Si l'élément est présent, supprimez-le; sinon, faites rien*
- Peu de collisions, ajout et suppression prend $O(1)$ temps

# ch16/hashtable/HashSet.java

```
1  import java.util.AbstractSet;
2  import java.util.Iterator;
3  import java.util.NoSuchElementException;
4
5  /**
6     A hash set stores an unordered collection of objects, using
7     a hash table.
8  */
9  public class HashSet extends AbstractSet
10 {
11    private Node[] buckets;
12    private int size;
13
14    /**
15       Constructs a hash table.
16       @param bucketsLength the length of the buckets array
17    */
18    public HashSet(int bucketsLength)
19    {
20       buckets = new Node[bucketsLength];
21       size = 0;
22    }
23
```

***Continued***

# ch16/hashtable/HashSet.java (cont.)

```
24      /**
25         Tests for set membership.
26         @param x an object
27         @return true if x is an element of this set
28      */
29      public boolean contains(Object x)
30      {
31         int h = x.hashCode();
32         if (h < 0) h = -h;
33         h = h % buckets.length;
34
35         Node current = buckets[h];
36         while (current != null)
37         {
38            if (current.data.equals(x)) return true;
39            current = current.next;
40         }
41         return false;
42      }
43
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
44      /**
45         Adds an element to this set.
46         @param x an object
47         @return true if x is a new object, false if x was
48         already in the set
49      */
50      public boolean add(Object x)
51      {
52         int h = x.hashCode();
53         if (h < 0) h = -h;
54         h = h % buckets.length;
55
56         Node current = buckets[h];
57         while (current != null)
58         {
59            if (current.data.equals(x))
60               return false; // Already in the set
61            current = current.next;
62         }
63         Node newNode = new Node();
64         newNode.data = x;
65         newNode.next = buckets[h];
66         buckets[h] = newNode;
67         size++;
68         return true;
69      }
70
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
71      /**
72         Removes an object from this set.
73         @param x an object
74         @return true if x was removed from this set, false
75         if x was not an element of this set
76      */
77      public boolean remove(Object x)
78      {
79         int h = x.hashCode();
80         if (h < 0) h = -h;
81         h = h % buckets.length;
82
83         Node current = buckets[h];
84         Node previous = null;
85         while (current != null)
86         {
87            if (current.data.equals(x))
88            {
89               if (previous == null) buckets[h] = current.next;
90               else previous.next = current.next;
91               size--;
92               return true;
93            }
94            previous = current;
95            current = current.next;
96         }
97         return false;
98      }
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
100       /**
101          Returns an iterator that traverses the elements of this set.
102          @return a hash set iterator
103       */
104       public Iterator iterator()
105       {
106          return new HashSetIterator();
107       }
108
109       /**
110          Gets the number of elements in this set.
111          @return the number of elements
112       */
113       public int size()
114       {
115          return size;
116       }
117
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
118     class Node
119     {
120        public Object data;
121        public Node next;
122     }
123
124     class HashSetIterator implements Iterator
125     {
126        private int bucket;
127        private Node current;
128        private int previousBucket;
129        private Node previous;
130
131        /**
132           Constructs a hash set iterator that points to the
133           first element of the hash set.
134        */
135        public HashSetIterator()
136        {
137           current = null;
138           bucket = -1;
139           previous = null;
140           previousBucket = -1;
141        }
142
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
143         public boolean hasNext()
144         {
145            if (current != null && current.next != null)
146               return true;
147            for (int b = bucket + 1; b < buckets.length; b++)
148               if (buckets[b] != null) return true;
149            return false;
150         }
151
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
152         public Object next()
153         {
154            previous = current;
155            previousBucket = bucket;
156            if (current == null || current.next == null)
157            {
158               // Move to next bucket
159               bucket++;
160
161               while (bucket < buckets.length
162                     && buckets[bucket] == null)
163                  bucket++;
164               if (bucket < buckets.length)
165                  current = buckets[bucket];
166               else
167                  throw new NoSuchElementException();
168            }
169            else // Move to next element in bucket
170               current = current.next;
171            return current.data;
172         }
173
```

*Continued*

# ch16/hashtable/HashSet.java (cont.)

```
174         public void remove()
175         {
176            if (previous != null && previous.next == current)
177               previous.next = current.next;
178            else if (previousBucket < bucket)
179               buckets[bucket] = current.next;
180            else
181               throw new IllegalStateException();
182            current = previous;
183            bucket = previousBucket;
184         }
185      }
186   }
```

# ch16/hashtable/HashSetDemo.java

```
1  import java.util.Iterator;
2  import java.util.Set;
3
4  /**
5     This program demonstrates the hash set class.
6  */
7  public class HashSetDemo
8  {
9     public static void main(String[] args)
10    {
11       Set names = new HashSet(101); // 101 is a prime
12
13       names.add("Harry");
14       names.add("Sue");
15       names.add("Nina");
16       names.add("Susannah");
17       names.add("Larry");
18       names.add("Eve");
19       names.add("Sarah");
20       names.add("Adam");
21       names.add("Tony");
22       names.add("Katherine");
23       names.add("Juliet");
```

# ch16/hashtable/HashSetDemo.java (cont.)

```
24          names.add("Romeo");
25          names.remove("Romeo");
26          names.remove("George");
27
28          Iterator iter = names.iterator();
29          while (iter.hasNext())
30             System.out.println(iter.next());
31       }
32    }
```

## Program Run:

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

# Calculer les codes de hachage

- La fonction de hachage calcule un entier à partir de l'objet
- Choisir une fonction de hachage de telle manière que les objets différents ont les codes de hachage plutôt différents
- Mauvaise choix pour une fonction de hachage des chaînes
  - *Additionner les valeurs d'encodage Unicode des caractères de la chaîne*

```
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = h + s.charAt(i);
```

  - *Permutations (`"eat«` et `"tea"`) auront le même code de hachage*

# Calculer les codes de hachage

- La fonction de hachage pour une chaîne `s` de la bibliothèque standard

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = HASH_MULTIPLIER * h + s.charAt(i)
```

- Par exemple, le code de hachage de *"eat"* est

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

- Le code de *"tea"* est assez différent:

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

# La méthode `hashCode` pour la classe `Coin`

- There are two instance fields: `String` coin name and `double` coin value
- Use `String`'s `hashCode` method to get a hash code for the name
- To compute a hash code for a floating-point number:
  - *Wrap the number into a `Double` object*
  - *Then use `Double`'s `hashCode` method*
- Combine the two hash codes using a prime number as the `HASH_MULTIPLIER`

# La méthode `hashCode` pour la classe `Coin`

```
class Coin
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(value).hashCode();
      final int HASH_MULTIPLIER = 29;
      int h = HASH_MULTIPLIER * h1 + h2;
      return h;
   }
   ...
}
```

# Création des codes de hachage pour vos classes

- Utilisez le nombre premier comme `HASH_MULTIPLIER`
- Calculez les codes de hachage pour chaque champ d'instance
- Pour un champ d'instance de type entier utilisez la valeur de ce champ
- Combinez les codes de hachage:

```
int h = HASH_MULTIPLIER * h1 + h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER *h + h4;
...
return h;
```

# Création des codes de hachage pour vos classes

- Votre méthode `hashCode` doit être compatible avec la méthode `equals`
  - *si* `x.equals(y)`, *donc* `x.hashCode() == y.hashCode()`
- Vous aurez de problèmes si votre classe définit la méthode `equals` mais pas la méthode `hashCode`
  - *Si on oubli de définir la méthode* `hashCode` *pour* `Coin`, *cette méthode sera héritée de la super classe* `Object`
  - *Cette méthode calcule un code de hachage à partir de l'adresse mémoire de l'objet*
  - *Effet: N'import quels deux objets auront fort probable les codes de hachage différents*

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

# Création des codes de hachage pour vos classes

- En général, définissez les deux méthodes , `hashCode` et `equals` ou rien

# Hash Maps

- Dans l'implémentation des cartes avec une table de hachage seulement les clés sont « hachées »
- Les clés nécessitent les méthodes `hashCode` et `equals` compatibles

# ch16/hashcode/Coin.java

```
1  /**
2     A coin with a monetary value.
3  */
4  public class Coin
5  {
6     private double value;
7     private String name;
8
9     /**
10       Constructs a coin.
11       @param aValue the monetary value of the coin.
12       @param aName the name of the coin
13    */
14    public Coin(double aValue, String aName)
15    {
16       value = aValue;
17       name = aName;
18    }
19
```

# ch16/hashcode/Coin.java (cont.)

```
20      /**
21         Gets the coin value.
22         @return the value
23      */
24      public double getValue()
25      {
26         return value;
27      }
28
29      /**
30         Gets the coin name.
31         @return the name
32      */
33      public String getName()
34      {
35         return name;
36      }
37
```

# ch16/hashcode/Coin.java (cont.)

```
38      public boolean equals(Object otherObject)
39      {
40         if (otherObject == null) return false;
41         if (getClass() != otherObject.getClass()) return false;
42         Coin other = (Coin) otherObject;
43         return value == other.value && name.equals(other.name);
44      }
45
46      public int hashCode()
47      {
48         int h1 = name.hashCode();
49         int h2 = new Double(value).hashCode();
50         final int HASH_MULTIPLIER = 29;
51         int h = HASH_MULTIPLIER * h1 + h2;
52         return h;
53      }
54
55      public String toString()
56      {
57         return "Coin[value=" + value + ",name=" + name + "]";
58      }
59   }
```

# ch16/hashcode/CoinHashCodePrinter.java

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  /**
5     A program that prints hash codes of coins.
6  */
7  public class CoinHashCodePrinter
8  {
9     public static void main(String[] args)
10    {
11       Coin coin1 = new Coin(0.25, "quarter");
12       Coin coin2 = new Coin(0.25, "quarter");
13       Coin coin3 = new Coin(0.05, "nickel");
14
15       System.out.println("hash code of coin1=" + coin1.hashCode());
16       System.out.println("hash code of coin2=" + coin2.hashCode());
17       System.out.println("hash code of coin3=" + coin3.hashCode());
18
19       Set<Coin> coins = new HashSet<Coin>();
20       coins.add(coin1);
21       coins.add(coin2);
22       coins.add(coin3);
23
```

# ch16/hashcode/CoinHashCodePrinter.java (cont.)

```
24          for (Coin c : coins)
25             System.out.println(c);
26       }
27    }
```

## Program Run:

```
hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-1768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name=nickel]
```