Things get sorted

Sorting

- We've looked at Arrays, ArrayLists and LinkedLists
- We're about to look at Sets
- Before we do, let's consider the case where we want to *sort* an Array or a List...

Sorting

- The Arrays class and Collections class in java supply sorting algorithms for arrays and collections
- This is straightforward for cases like sorting an array of integers:

```
int[] intArr = { 5, 4, 3, 2, 1 };

Arrays.sort(intArr);

for (int num : intArr) {

System.out.println(num);
```

```
>
1
2
3
4
5
```

Sorting Objects

- Seems easy enough, but...
- What if we want to sort an array of Employees, or Bikes, or Accounts?
- Sorting integers is easy, but how does Java know how to sort these complex objects?
- We have to *tell Java* how to sort them

Comparable

- We do that by having our class implement the Comparable interface
- The Comparable interface lists a single method compareTo()
- We have our class implement that method in a way that tells Java how objects of the class should be sorted

The `Comparable` Interface

- The `Comparable` interface is declared as follows:

```
public interface Comparable<T> {
   int compareTo(T other)
}
```

  - the integer returned by `a.compareTo(b)` must adhere to the following convention:
    - negative if a < b
    - zero if `a.equals(b)`
    - positive if a > b

- `compareTo` defines the ***natural ordering*** for the class

Golfers

- Let's say we have a class Golfer
- It might have attributes and methods like this:

```
public class Golfer {
private int handicap;
private int bestscore;
private String name;

public Golfer(int hand, int best, String name)
{
bestscore = best;
handicap = hand;
this.name = name;
}
```

continued

Golfer

```
public int getHand()
{
return handicap;
}

public int getBest()
{
return bestscore;
}

public String getName()
{
return name;
}
```

Golfer

- We want to create a sorted Array of Golfers, and we want them sorted by handicap
- We need to tell Java that they should be sorted this way

Golfers

- Let's say we have a class Golfer
- It might have attributes and methods like this:

```
public class Golfer implements Comparable<Golfer> {
private int handicap;
private int bestscore;
private String name;

public Golfer(int hand, int best, String name)
{
bestscore = best;
handicap = hand;
this.name = name;
}
```

We make our class implement Comparable

continued

Golfers

- Let's say we have a class Golfer
- It might have attributes and methods like this:

```
public class Golfer implements Comparable<Golfer> {
private int handicap;
private int bestscore;
private String name;

public Golfer(int hand, int best, String name)
{
bestscore = best;
handicap = hand;
this.name = name;
}
```

And if we implement Comparable, then we need to define the compareTo method

continued

Comparing Golfers

- We add this method to our Golfer class

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return -1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return 1;
}
```

Comparing Golfers

- We add this method to our Golfer class

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return -1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return 1;
}
```

So when compareTo gets called on a Golfer object, with another Golfer passed as a parameter, we compare their handicaps and return -1, 0 or 1 indicating how they should be sorted

Comparing Golfers

- We add this method to our Golfer class

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return -1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return 1;
}
```

Now when we put a bunch of Golfers in an array and call Arrays.sort(), Java will use the compareTo method to put each in its correct place

Comparing Golfers

```
Golfer[] golfBuddies = new Golfer[3];
Golfer bob = new Golfer(9, 85, "bob");
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
golfBuddies[0] = bob;
golfBuddies[1] = jane;
golfBuddies[2] = jim;
Arrays.sort(golfBuddies);
for (Golfer g: golfBuddies)
{
System.out.println(g.getName());
}
```

Comparing Golfers

```
Golfer[] golfBuddies = new Golfer[3];
Golfer bob = new Golfer(9, 85, "bob");
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
golfBuddies[0] = bob;
golfBuddies[1] = jane;
golfBuddies[2] = jim;
Arrays.sort(golfBuddies);
for (Golfer g: golfBuddies)
{
System.out.println(g.getName());
}
```

```
>
jane
bob
jim
```

Comparing Golfers

- How could we change the compareTo method to rank them in reverse order?

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return -1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return 1;
}
```

Comparing Golfers

- How could we change the compareTo method to rank them in reverse order?

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap > ohand)
{
return -1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return 1;
}
```

We could change the `if` statement

Comparing Golfers

- How could we change the compareTo method to rank them in reverse order?

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return 1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return -1;
}
```

Or instead we could change the return values

Comparing Golfers

- How could we change the compareTo method to rank them in reverse order?

```
public int compareTo(Golfer g)
{
int ohand = g.getHand();
if (this.handicap < ohand)
{
return 1;
}
else if (this.handicap == ohand)
{
return 0;
}
else return -1;
}
```

Either way:

```
>
jim
bob
jane
```

compareTo

- When a.compareTo(b) is called, the return values mean:
  - -1 a comes before b
  - 1 b comes before a
  - 0 they are equal

Comparing Golfers

- What if we decided we want to rank Golfers by their best score instead?
- We could simply define compareTo() differently

Comparing Golfers

```
public int compareTo(Golfer g)
{
int obest = g.getBest();
if (this.best < obest)
{
return -1;
}
else if (this.best == obest)
{
return 0;
}
else return 1;
}
```

Implementing compareTo

- So if we decide we want to sort objects based on a different attribute, do we need to rewrite compareTo?
- Some objects might have a single natural ordering, but others may have many attributes and sometimes we want to sort by one and sometimes by another
- Also, what if we want to sort objects of a class defined by someone else? In that case, we can't go in and redefine their compareTo method

Comparators

- There is an alternative to implementing Comparable and its compareTo method
- We can use *Comparators*

The `Comparator` Interface

- A comparator has to implement:

```
public interface Comparator<T> {
    int compare(T object1, T object2);
}
```

The `Comparator` Interface

- The return value for this method
  - is defined in the same way as for the `compareTo` method of the `Comparable` interface:

compare(a,b) is like *a.compareTo(b)*

- We may define many comparators for a class if we need to order objects of that type in different ways.

Golf Comparators

- We could define two different Golfer comparators:

```
public class HandicapComparator implements
Comparator<Golfer>{


public int compare(Golfer g1, Golfer g2)

{

   return g1.getHand() - g2.getHand();

}

}
```

Golf Comparators

- And other for best scores:

```
public class BestScoreComparator implements
Comparator<Golfer>{


public int compare(Golfer g1, Golfer g2)

{

   return g1.getBest() - g2.getBest();

}

}
```

Golf Comparators

- And other for best scores:

```
public class BestScoreComparator implements
Comparator<Golfer>{


public int compare(Golfer g1, Golfer g2)

{

   return g1.getBest() - g2.getBest();

}

}
```

What's going on here? Don't we need to return -1, 0 or 1? This isn't guaranteed to do so.

Golf Comparators

- And other for best scores:

```
public class BestScoreComparator implements
Comparator<Golfer>{


public int compare(Golfer g1, Golfer g2)

{

   return g1.getBest() - g2.getBest();

}

}
```

Actually, no. We just need to return a negative value, positive value, or zero. Often people use -1, 0 and 1, but not always. For that reason, be careful doing something like `if (this.compareTo(g1) == -1)`because it may be a different negative value

Using Comparators

- So how do we use these Comparator classes?
- Methods such as sort() are overloaded
  - If the class implements Comparable and has a compareTo method, we just call sort() on our array or collection
  - There is also a version of sort() that takes a Comparator as a second parameter
- We can take one of those two approaches
- Note: if the class implements Comparable ***and*** we pass a Comparator, it is the Comparator that gets used for sorting

Using Comparators

```
Golfer[] golfBuddies = new Golfer[3];
Golfer bob = new Golfer(9, 85, "bob");
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
golfBuddies[0] = bob;
golfBuddies[1] = jane;
golfBuddies[2] = jim;
Arrays.sort(golfBuddies, new HandicapComparator());
for (Golfer g: golfBuddies)
{
System.out.println(g.getName());
}
```

Now we can just provide whatever Comparator we want

In-Class Exercise I

- Change this class Worker so that it implements Comparable

```
public class Worker {
private int id;
private int age;

public Worker(int anID, int anAge)
 {
  this.id = anID;
  this.age = anAge;
 }
}
```

In-Class Exercise I

- Now write two Comparator classes to give us more flexibility on how we sort Workers

```
public class Worker {
private int id;
private int age;

public Worker(int anID, int anAge)
 {
  this.id = anID;
  this.age = anAge;
 }
```

```
}
```

Sorting

- We'll come back to Comparable and Comparator shortly, and in more detail, when we talk about Sets

The `Set` Interface

You will be expected to:

• program to the generic Set and SortedSet interfaces including read and use the API's
• justify the use of a set vs a list for a given problem
• compare and contrast the HashSet and TreeSet classes (benefits of using each, basic run time analysis)
• design and implement a class in such a way that it can be used with the Java collections framework
- overrides equals and hashCode
- implements the generic Comparable and Comparator interfaces to account for multiple sorting criteria

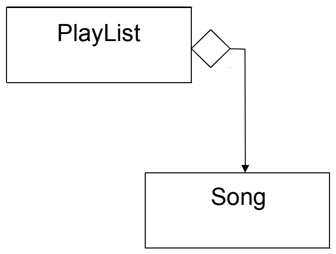
Reading:

- 2nd Ed: 19.8, 21.1, briefly: 21.3, 21.4
- 3rd/4th Ed: 14.8, 16.1, briefly: 16.3, 16.4

Using a Set

• A playlist is a set of songs:

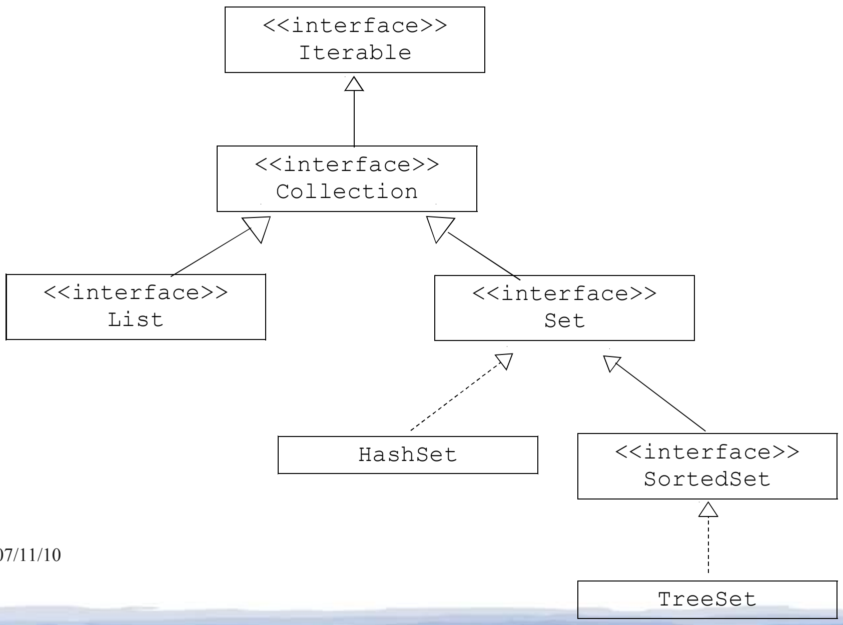
Chris's Play List
Meet Me Halfway
Jingle Bells
21 Guns
Thriller



Using a Set (continued)

- In a `List`,
  - Each object has a position
  - We can put the same object in the list multiple times
- Sometimes, we need the functionality of a mathematical set
  - No duplicates in the set
  - Members do not have a position in the set
- For example…
  - `MenuItems` that appear in the `Menu` of a restaurant
  - `Songs` that appear on a `PlayList`
  - `Student` enrolled in a `Course`
- In these cases we use a `Set` not a `List`

The `Set` interface



Methods of the `Set` interface

- Note that the `Set` interface extends the `Collection` interface. An implementation of `Set` therefore supports the methods defined in the `Collection` interface:
  - `add(o)` – add a specified element to the set (if not already a member)
  - `remove(o)` – remove the specified element from the set
  - `contains(o)` – is the specified element in the set?
  - etc.
- Note that the add method:

```
  public boolean add( E item );
```

  adds the item only if it isn't already in the set. The method returns true if the item is added and false if it's already in the set.
- Similarly the `addAll` method does not add duplicates.

Methods of the `Set` interface

- We can use these methods to define known set operations:
  - `c1.containsAll(c2)` - true if `c2` is a subset of `c1`
  - `c1.addAll(c2)` - `c1` becomes union of c1 and c2
  - `c1.retainAll(c2)` - `c1` becomes intersection of c1 and c2
  - `c1.removeAll(c2)` - `c1` becomes set difference of c1 and c2

Using Sets

- Since Set is specified as an interface, to use it we have to pick a particular implementation (e.g., `HashSet`, or `TreeSet`)
- Example:

```
public class PlayList
{
private Set<Song> songs;

public PlayList()
{
        songs= new HashSet< Song >();
}
```

The `HashSet` implementation

- The `HashSet` implementation provides an efficient implementation of the `Set` interface that allows us to add or remove an item or check if the set contains an item in O(1) time provided certain conditions are met (more later).

- That is, if

```
Set<…>  s = new HashSet<…>();
```

  - `s.add(o)` is O(1)
  - `s.remove(o)` is O(1)
  - `s.contains(o)` is O(1)

The `HashSet` implementation

- As mentioned, certain conditions must be met if we are to add, remove or determine if the set contains an item in O(1) time.
- To understand these conditions, we must have a basic understanding of how the hash set works.
- A hash set uses a ***hash table*** as the underlying structure in which data is stored.
- A hash table is an array of linked lists…

The `HashSet` implementation

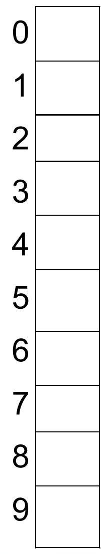
- We add elements to the table using a hash code, an integer that represents the object
- A hash set maintains a list of groups.
- All members of the group at position i have a hashCode of i.
- We'll talk more in a moment about where these hashCodes come from
- Let's see an example….

Hash Table Example

- Suppose we want to add integers to a hash table using the following hash code:

```
hashCode = value%10;
```

- What does the table look like after inserting:
  243556,
  329394,
  3348,
  436,
  3234,
  424

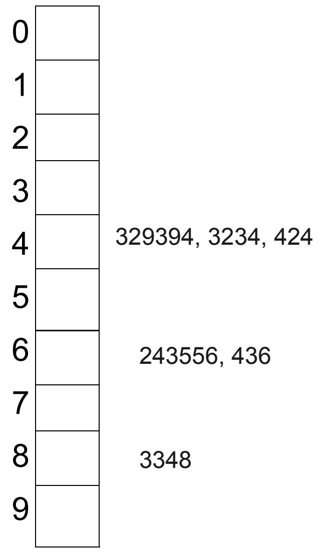


Hash Table Example

- Suppose we want to add integers to a hash table using the following hash code:

```
hashCode = value%10;
```

- What does the table look like after inserting:
  243556,
  329394,
  3348,
  436,
  3234,
  424



Hash Tables

- Hashing can be used to find elements in a data structure quickly without making a linear search
- A *hash table* can be used to implement sets and maps
- A *hash function* computes an integer value (called the *hash code*) from an object
- A good hash function minimizes *collisions* – identical hash codes for different objects
- To compute the hash code of object `x`:

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

Collisions

- Notice in the previous example that we had quite a few collisions – items that are stored in the same location (or bucket)
- We want a good hash code that will reduce these collisions

Sample Strings and Their Hash Codes

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

Sample Strings and Their Hash Codes

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

Note: the String class has an already defined hashCode method we can use

Simplistic Implementation of a Hash Table

- To implement
  - *Generate hash codes for objects*
  - *Make an array*
  - *Insert each object at the location of its hash code*
- To test if an object is contained in the set
  - *Compute its hash code*
  - *Check if the array position with that hash code is already occupied*

Problems with Simplistic Implementation

- It is not possible to allocate an array that is large enough to hold all possible integer index positions
- It is possible for two different objects to have the same hash code

Solutions

- Pick a reasonable array size and reduce the hash codes to fall inside the array

```
int h = x.hashCode();
if (h < 0) h = -h;
h = h % size;
```

- When elements have the same hash code:
  - *Use a node sequence to store multiple objects in the same array position*
  - *These node sequences are called buckets*

Buckets

- So instead of a single object being stored at each point in the array, we have a LinkedList of objects at each point
- This allows for the possibility that some different objects will have the same hash code by chance and thus be stored at the same array index
- We say they are in the same bucket

Hash Table with Buckets to Store Elements with Same Hash Code

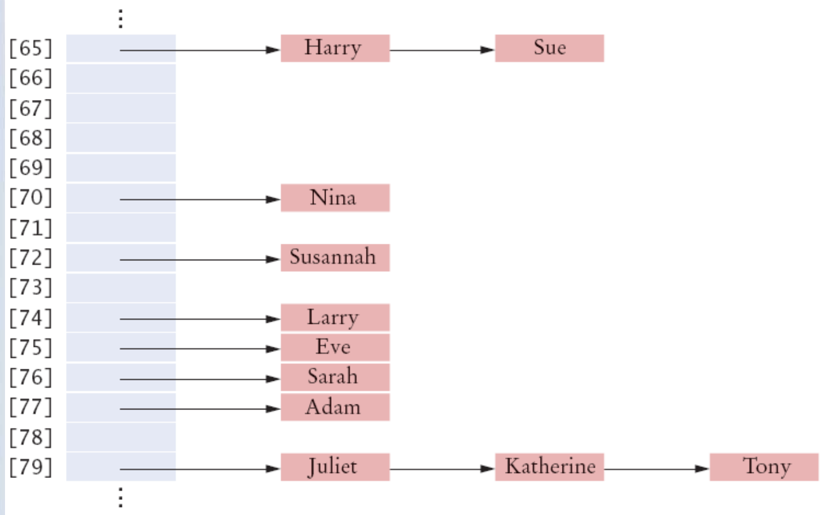


Figure 6 A Hash Table with Buckets to Store Elements with the Same Hash Code

Algorithm for Finding an Object `x` in a Hash Table

1. Get the index h into the hash table
   - *Compute the hash code*
   - *Reduce it modulo the table size*
2. Iterate through the elements of the bucket at position `h`
   - *For each element of the bucket, check whether it is equal to x*
3. If a match is found among the elements of that bucket, then `x` is in the set
   - *Otherwise, `x` is not in the set*

Hash Tables

- A hash table can be implemented as an array of buckets
- Buckets are sequences of nodes that hold elements with the same hash code
- If there are few collisions, then adding, locating, and removing hash table elements takes constant time
  - *Big-Oh notation: O(1)*
- For this algorithm to be effective, the bucket sizes must be small
- The table size should be a prime number larger than the expected number of elements
  - *An excess capacity of 30% is typically recommended*

Hash Tables

- Adding an element: simple extension of the algorithm for finding an object
  - *Compute the hash code to locate the bucket in which the element should be inserted*
  - *Try finding the object in that bucket*
  - *If it is already present, do nothing; otherwise, insert it*
- Removing an element is equally simple
  - *Compute the hash code to locate the bucket in which the element should be inserted*
  - *Try finding the object in that bucket*
  - *If it is present, remove it; otherwise, do nothing*
- If there are few collisions, adding or removing takes *O*(1) time

The `HashSet` implementation

- In a HashSet the operations are performed as following:

  `add(o)`
  - compute the hashcode of o, say i
  - add o in the ith group

  `remove(o)`
  - compute the hashcode of o, say i
  - search the ith group and remove o

  `contains(o)`
  - compute the hashcode of o, say i
  - search the ith group to find o
- If each group is small (and of constant size) each of the above operations is O(1).

The `HashSet` Implementation

- What makes these operations so efficient?
  - Take add() for example
- Rather than iterating over a collection and checking at each step whether the object already exists, we just compute the hashCode and check that index in the array
- We then check whether the object exists in that bucket
- If we have a good hashCode and hash table, there will be few collisions, meaning few items to search through in the bucket
- If we can get close to 1 item per bucket, these operations will be 0(1) – constant time

Where do hash codes come from?

- Each Java class inherits a hashCode() method from the Java class Object
  - when invoked, hashCode() returns an integer that represents the object
  - a class' hashCode() is usually defined in terms of the hash codes of its attributes
  - if two objects are equal according to equals(), they must have the same hash code
  - objects with the same hash code are not necessarily equal

- It would be nice to rely upon the Java Object's class definition of hashCode() but you can't if you override equals() because two instances of an object that are equal according to equals() may not return the same hashCode() unless you ensure they do!

- The rule is then:
  "If you override equals() you should always override hashCode()"
  - See page 36 of http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf for a complete description

Default hashCode() and equals()

- If you rely on the default inherited equals() and hashCode(), you are okay in the sense that they both rely on the memory location of the object and are therefore consistent with one another
- But then you are left with a very restrictive definition of equals() which might not be what you want

How do you write a good hashCode()

- Writing a fantastic hashCode() method for a class is hard
  - ➢The kind of thing people write PhD theses about
- Writing a decent hashCode() method for a class is straightforward
  - Page 38 of http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf provides a recipe.
    - Start with a non-zero value (preferably a prime number, like 11, 17, etc.) in the result value
    - Pick another prime number, say 37, as a multiplier
    - For each attribute that is taken into account in the equals() method
      - if attribute is of a primitive type (i.e. an integer, float, etc.) ,
        result = 37 * result + attribute's value casted to an integer
      - if attribute is an object,
        result = 37 * result + attribute.hashCode()
      - and so on…

`Song` Example

```
public class Song{
   private String title;
   private Artist artist;
   private int lengthInSeconds;
   private Album album;
   private int playCount;
```

Song Example (cont.)

```
public boolean equals( Object other ){
   if  (other == null)
      return false;
   if( getClass() != other.getClass() )
      return false;
   Song otherItem = (Song) other;

  return(title.equals(other.title) &&
artist.equals(other.artist));

  }
```

Song Example (cont.)

```
public int hashCode() {

 int result = 17;
 final int MULT = 31;
 result = MULT*result + title.hashCode();
 result = MULT*result + artist.hashCode();
 return result;

 }
. . .
} // end Song
```

Computing `Hash` Codes

- A hash function computes an integer hash code from an object
- Choose a hash function so that different objects are likely to have different hash codes.
- Bad choice for hash function for a string
  - *Adding the unicode values of the characters in the string*

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

  - *Because permutations ("eat" and "tea") would have the same hash code*

Computing Hash Codes

- Hash function for a string s from standard library

-
  ```
  final int HASH_MULTIPLIER = 31;
  int h = 0;
  for (int i = 0; i < s.length(); i++)
     h = HASH_MULTIPLIER * h + s.charAt(i)
  ```
- For example, the hash code of "eat" is

  `31 * (31 * 'e' + 'a') + 't' = 100184`
- The hash code of "tea" is quite different, namely

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

A `hashCode` Method for the Coin Class

- 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`

A hashCode Method for the Coin Class

```
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;
   }
   . . .
}
```

Creating Hash Codes for your Classes

Use a prime number as the `HASH_MULTIPLIER`

Compute the hash codes of each instance field

For an integer instance field just use the field value

Combine the hash codes

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

Creating Hash Codes for your Classes

- Your `hashCode` method must be compatible with the `equals` method
  - *if x.equals(y) then x.hashCode() == y.hashCode()*
- You get into trouble if your class defines an `equals` method but not a `hashCode` method
  - *If we forget to define `hashCode` method for `Coin` it inherits the method from `Object` superclass*
  - *That method computes a hash code from the memory location of the object*
  - *Effect: any two objects are very likely to have a different hash code*

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

- In general, define either both `hashCode` and equals methods or neither

The `SortedSet` Interface

- Allows the user to retrieve objects from the set in sorted order
- To sort a collection, the objects within the collection must be comparable:
  - the corresponding class must implement either the `Comparable` interface or the `Comparator` interface.

The `Comparable` Interface

- The `Comparable` interface is declared as follows:

```
public interface Comparable<T> {
   int compareTo(T other)
}
```

  - the integer returned by `a.compareTo(b)` must adhere to the following convention:
    - negative if a < b
    - zero if `a.equals(b)`
    - positive if a > b
- `compareTo` defines the ***natural ordering*** for the class

Implementing `compareTo`

- Rules to follow when you implement this method in a class `C`:
  - `C` must implement `Comparable<C>`
  - must be asymmetric
    - `a.compareTo(b)` and `b.compareTo(a)` must both equal 0 or have opposite signs
  - must be transitive
    - if `a.compareTo(b) < 0` and `b.compareTo(c) < 0` then `a.compareTo(c) < 0`
  - must be consistent with `equals()`
    - `a.equals(b)` is true iff `a.compareTo(b)` is zero and `b.compareTo(a)` is zero

The `Comparator` Interface

- Some classes may not have a single natural ordering
  - employees may be ordered by name or by salary or…
- A comparator is an object that defines (encapsulates) one ordering for a class
- A comparator has to implement:

```
public interface Comparator<T> {
    int compare(T object1, T object2);
}
```

The `Comparator` Interface

- The return value for this method
  - is defined in the same way as for the `compareTo` method of the `Comparable` interface:

compare(a,b) is like *a.compareTo(b)*

- We may define many comparators for a class if we need to order objects of that type in different ways.

Example

- Create a `Comparator` that compares `Account`s by id numbers.

```
public class AccountIdComparator
              implements Comparator<Account>
{
   public int compare( Account ac1, Account ac2 )
   {
      return (ac1.getId() - ac2.getId() );
   }
}
```

The `SortedSet` Interface

```
public interface SortedSet<E> extends Set<E>
{
  // Views on the sorted set
  SortedSet<E>  subSet(E from, E to);
  SortedSet<E>  headSet(E toElement);
  SortedSet<E>  tailSet(E fromElement);

  // Endpoints
  E first();
  E last();

  // Comparator access
  Comparator<? super E>  comparator();
}
```

The `SortedSet` Interface

- Like `Set` but keeps elements in ascending order according to
  - the *natural order* defined by the `compareTo` method of `Comparable`, or
  - the `compare` method of a `Comparator`
- Iterator will traverse elements in the defined order
- Array produced by `toArray` methods is sorted
- Additional operations:
  - `first()` and `last()` return min and max elements in set
  - `comparator()` returns the `Comparator` used to sort the set, or `null` if the *natural order* is used

The `TreeSet` Class

- The `TreeSet` class implements the `SortedSet` interface. It has the following constructors (among others):

```
public TreeSet()
// orders the elements according to their
// natural order


public TreeSet( Comparator< ? super E > c )
// orders the elements according to the
// comparator c
```

The `TreeSet` Class

- Note the use of the bounded wildcard:
  - Comparator< ? super E > c
- This indicates that the `Comparator` must compare types that are supertypes of `E` (including `E` itself).
- For example, if `SavingsAccount` is a subclass of `Account` and `BalanceComparator` implements the `Comparator<Account>` interface, then we can create the following `TreeSet` of `SavingsAccount` objects:

```
TreeSet<SavingsAccount> accts
      = new TreeSet<SavingsAcount>(
                          new BalanceComparator() );
```

TreeSet - Time Complexity

- The add, remove and contains methods all have a guaranteed O(log N) time complexity.
- So these operations on a `TreeSet` are less efficient than for a `HashSet` (assuming a good `hashCode()` implementation) but we have to remember that the `TreeSet` maintains the data in sorted order.

TreeSet vs. HashSet

- If you don't care about sorting but just want efficient add(), remove() and contains() operations, the question of which Set to use depends on how confident you are in your hash code method
- If you have a good hash code, there will be few collisions, which means few objects in each bucket, which means less to search through
- Otherwise, you might want to use a TreeSet

Using TreeSet

- Now we can do this:

```
Set<Golfer> gSet = new TreeSet<Golfer>();
gSet.add(bob);
gSet.add(jane);
gSet.add(jim);
Iterator<Golfer> itr = gSet.iterator();
while (itr.hasNext())
{
System.out.println(itr.next().getName());
}
```

Using TreeSet

- Or we can supply a Comparator

```
Set<Golfer> gSet = new TreeSet<Golfer>(new
HandicapComparator());
gSet.add(bob);
gSet.add(jane);
gSet.add(jim);
Iterator<Golfer> itr = gSet.iterator();
while (itr.hasNext())
{
System.out.println(itr.next().getName());
}
```

Using TreeSet

- A different Comparator if we choose...

```
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());

gSet.add(bob);

gSet.add(jane);

gSet.add(jim);

Iterator<Golfer> itr = gSet.iterator();

while (itr.hasNext())

{

System.out.println(itr.next().getName());

}
```

Using TreeSet

- Now Java will use either the compareTo() method if we implement Comparable, or the compare() method if we use Comparators, and will keep our items nicely sorted
- Whenever we add something, Java will determine where it belongs by calling those methods
- Note: if we don't supply a Comparator and our class doesn't implement Comparable, we will get an error. We need one or the other.

Exercises

- More Exercises:
  - 2nd Ed: P19.12, P21.1, P21.11 (but use `HashSet<Integer>` rather than their `IntSet` class)
  - 3rd Ed: P14.12, P16.1, P16.12 (but use `HashSet<Integer>` rather than their `IntSet` class)

In-Class Exercise II

- Write the equals() and hashCode() methods for our Golfer class

```
public class Golfer {
private int handicap;
private int bestscore;
private String name;

public Golfer(int hand, int best, String name)
{
bestscore = best;
handicap = hand;
this.name = name;
}
```

Learning Goals Review

You will be expected to:

• program to the generic Set and SortedSet interfaces including read and use the API's
• justify the use of a set vs a list for a given problem
• compare and contrast the HashSet and TreeSet classes (benefits of using each, basic run time analysis)
• design and implement a class in such a way that it can be used with the Java collections framework

- overrides equals and hashCode
- implements the generic Comparable and Comparator interfaces to account for multiple sorting criteria