Contents

- An Overview of the Java Collections Framework
- Linked Lists
- Sets
- Maps

In this chapter, you will learn about the Java collection framework, a hierarchy of interface types and classes for collecting objects

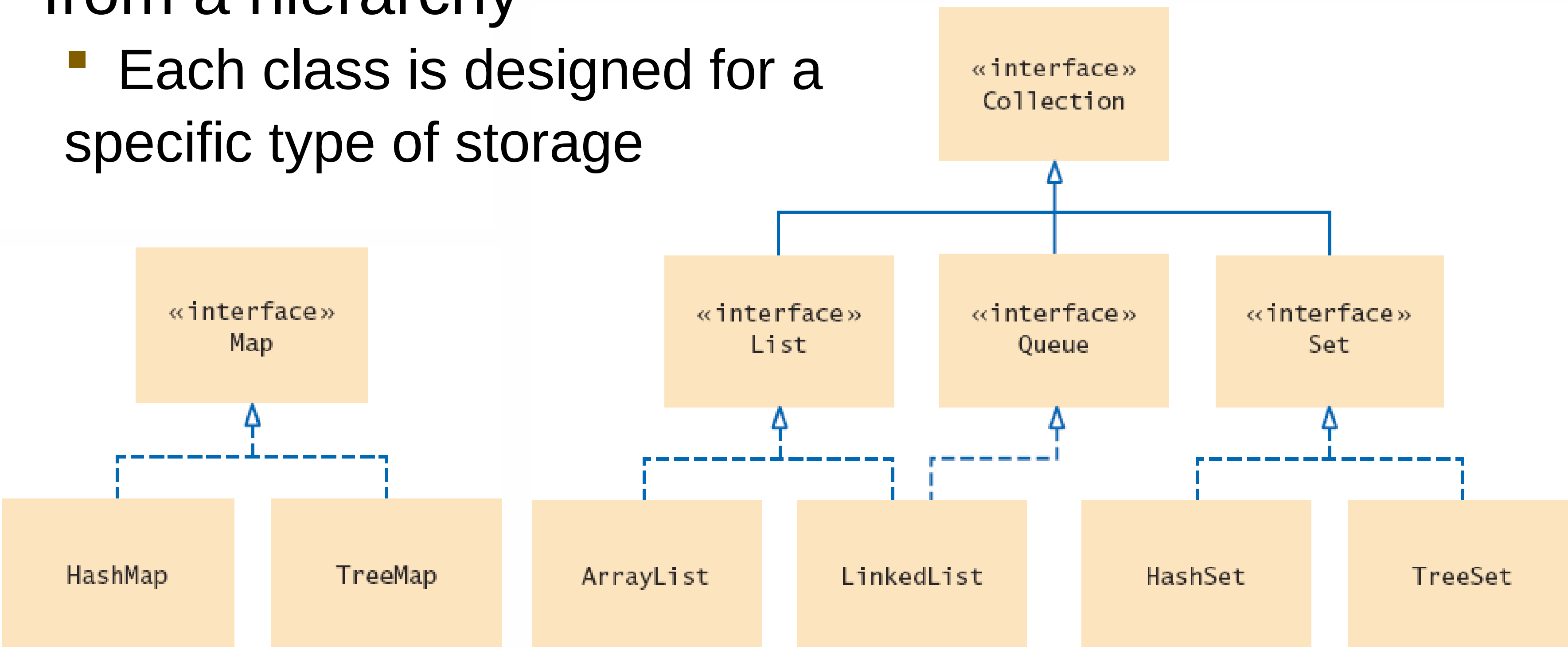
Java Collections Framework

- When you need to organize multiple objects in your program, you can place them into a collection
- For example, the `ArrayList` class is one of many collection classes that the standard Java library supplies
- Each interface type is implemented by one or more classes

A collection groups together elements and allows them to be accessed and retrieved later

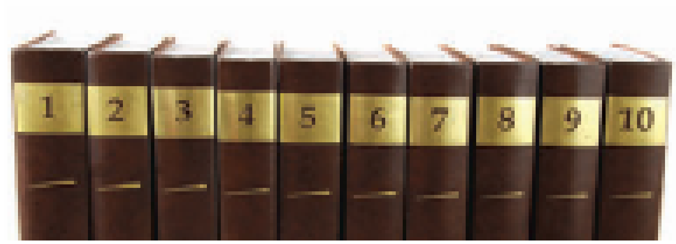
Collections Framework Diagram

- Each collection class implements an interface from a hierarchy
  - Each class is designed for a specific type of storage



Lists and Sets

- Ordered Lists



  - `ArrayList`
    - Stores a list of items in a dynamically sized array
  - `LinkedList`
    - Allows speedy insertion and removal of items from the list

A **list** is a collection that maintains the order of its elements.

Lists and Sets

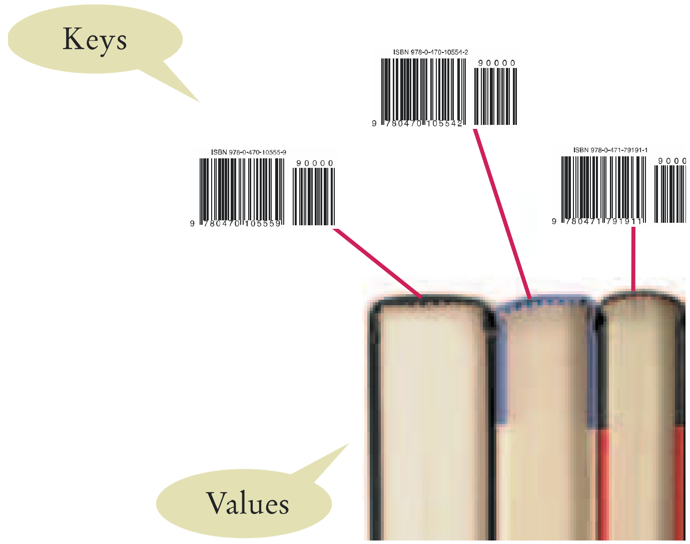
- Unordered Sets

  - `HashSet`
    - Uses hash tables to speed up finding, adding, and removing elements
  - `TreeSet`
    - Uses a binary tree to speed up finding, adding, and removing elements

A **set** is an unordered collection of unique elements.

Maps

- A map stores keys, values, and the associations between them
  - Example:
  - Barcode keys and books



A map keeps associations between key and value objects.

- Keys
  - Provides an easy way to represent an object (such as a numeric bar code)
- Values
  - The actual object that is associated with the key

The Collection Interface (1)

- List, Queue and Set are specialized interfaces that inherit from the Collection interface
  - All share the following commonly used methods

Table 1 The Methods of the Collection Interface

Collection<String> coll = new ArrayList<String>();	The ArrayList class implements the Collection interface.
coll = new TreeSet<String>()	The TreeSet class (Section 15.3) also implements the Collection interface.
int n = coll.size();	Gets the size of the collection. n is now 0.
coll.add("Harry"); coll.add("Sally");	Adds elements to the collection.
String s = coll.toString();	Returns a string with all elements in the collection. s is now "[Harry, Sally]"
System.out.println(coll);	Invokes the toString method and prints [Harry, Sally].

The `Collection` Interface (2)

Table 1 The Methods of the `Collection` Interface

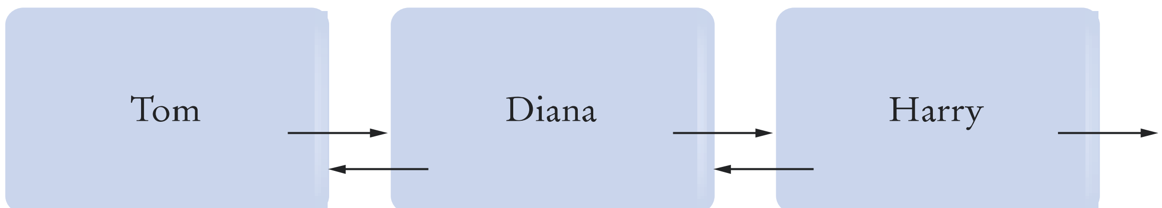
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`   System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

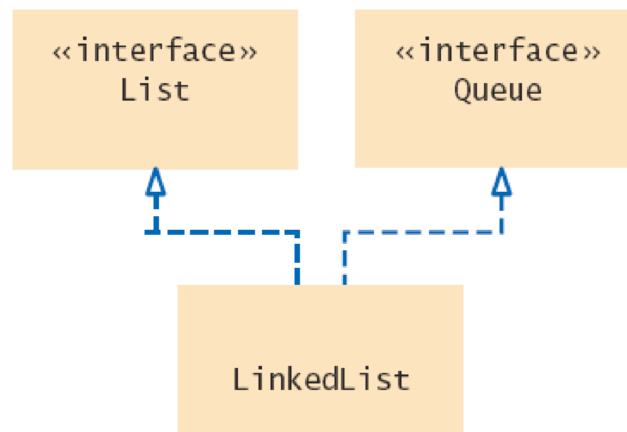
`Iterator<String> iter = coll.iterator()`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

Linked Lists

- Linked lists use references to maintain an ordered lists of ‘nodes’
  - The ‘head’ of the list references the first node
  - Each node has a value and a reference to the next node

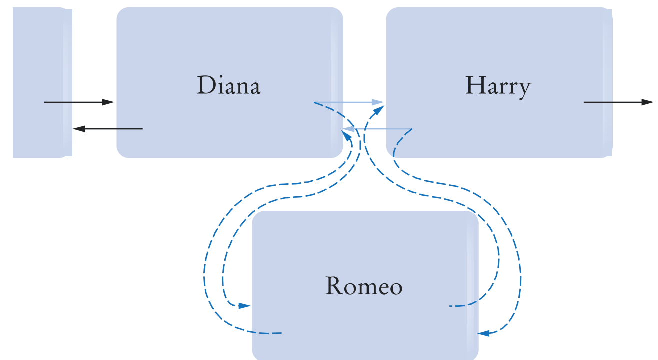


  - They can be used to implement
    - A `List` Interface
    - A `Queue` Interface

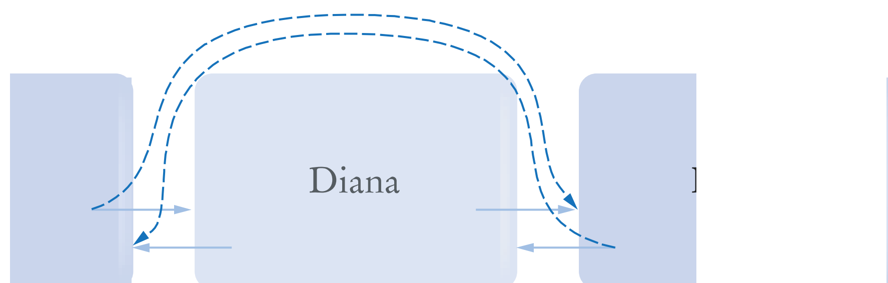


Linked Lists Operations

- Efficient Operations
  - Insertion of a node
    - Find the elements it goes between
    - Remap the references



  - Removal of a node
    - Find the element to remove
    - Remap neighbor’s references



  - Visiting all elements in order
- Inefficient Operations
  - Random access

Each instance variable is declared just like other variables we have used.

`LinkedList`: Important Methods

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<String>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 676).

Generic Linked Lists

- The Collection Framework uses Generics
  - Each list is declared with a type field in < > angle brackets

```
LinkedList<String> employeeNames = . . .;
```

```
LinkedList<String>
LinkedList<Employee>
```

List Iterators

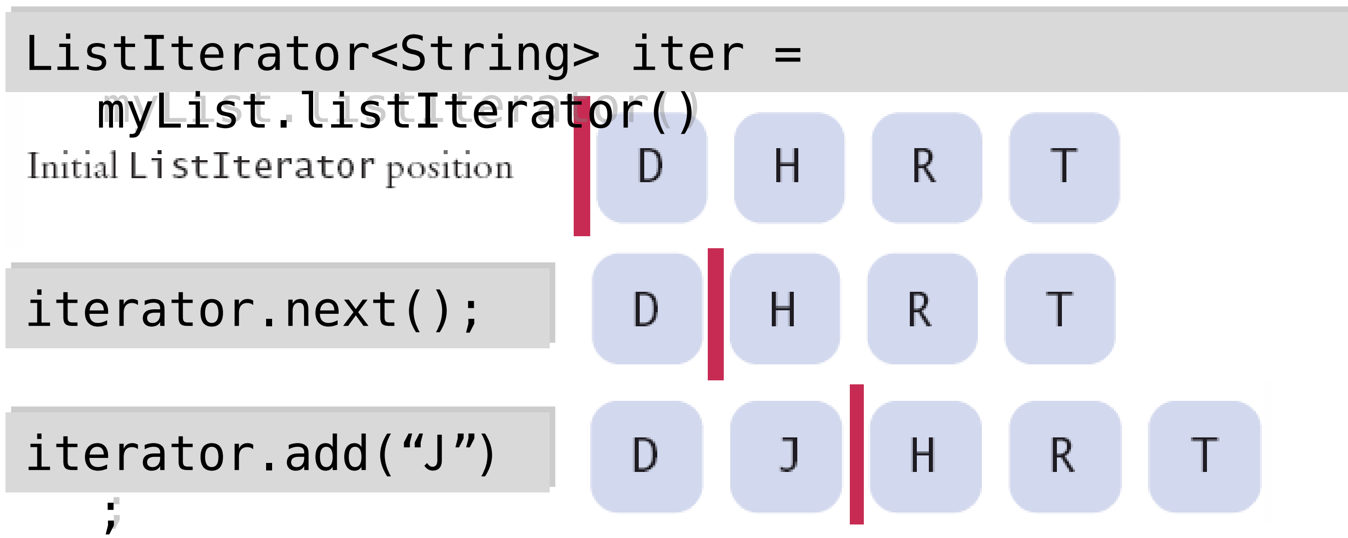
- When traversing a `LinkedList`, use a `ListIterator`
  - Keeps track of where you are in the list.

- Use an iterator to:
  - Access elements inside a linked list
  - Visit other than the first and the last nodes

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iter = employeeNames.listIterator()
```

Using Iterators

- Think of an iterator as pointing **between** two elements



- Note that the generic type for the `listIterator` must match the generic type of the `LinkedList`

Iterator and ListIterator Methods

- Iterators allow you to move through a list easily
  - Similar to an index variable for an array

Table 3 Methods of the `Iterator` and `ListIterator` Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.previous();` `iter.set("Juliet");`	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Iterators and Loops

- Iterators are often used in while and “for-each” loops
  - `hasNext` returns true if there is a next element
  - `next` returns a reference to the value of the next element

```
while (iterator.hasNext())
{
    String name =
   iterator.next();
    // Do something with nam
}
```

```
for (String name :
   employeeNames)
{
    // Do something with name
}
```

  - Where is the iterator in the “for-next” loop?
    - It is used ‘behind the scenes’

Adding and Removing with Iterators

- Adding `iterator.add("Juliet");`
  - A new node is added AFTER the Iterator
  - The Iterator is moved past the new node
- Removing
  - Removes the object that was returned with the last call to `next` or `previous`
  - It can be called only once after `next` or `previous`
  - You cannot call it immediately after a call to `add`.

If you call the `remove` method improperly, it throws an `IllegalStateException`.

```
while (iterator.hasNext())
{
  String name = iterator.next();
  if (condition is true for
   name)
  {
    iterator.remove();
  }
}
```

ListDemo.java (1)

- Illustrates adding, removing and printing a list

```java
import java.util.LinkedList;
import java.util.ListIterator;

/**
   This program demonstrates the LinkedList class.
*/
public class ListDemo
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Diana");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT

```

ListDemo.java (2)

```java
        // Add more elements after second element

        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT

        iterator.next(); // DHJNR|T

        // Remove last traversed element

        iterator.remove(); // DHJN|T

        // Print all elements

        System.out.println(staff);
        System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
    }
}
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

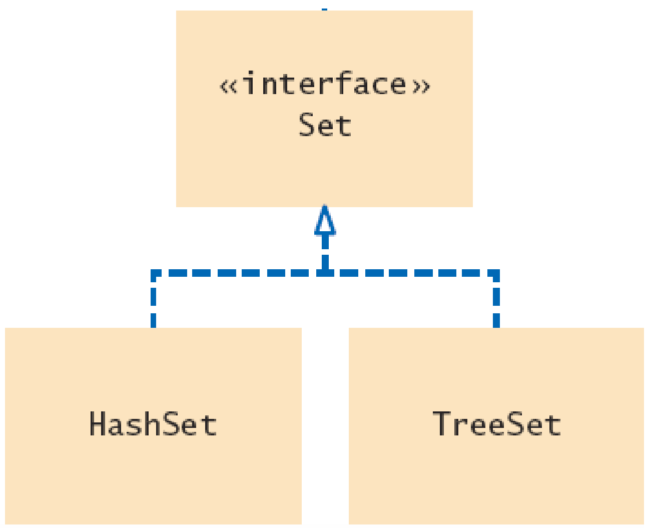
Sets

- A set is an unordered collection
  - It does not support duplicate elements
- The collection does not keep track of the order in which elements have been added
  - Therefore, it can carry out its operations more efficiently than an ordered collection

The `HashSet` and `TreeSet` classes both implement the Set interface.

Sets

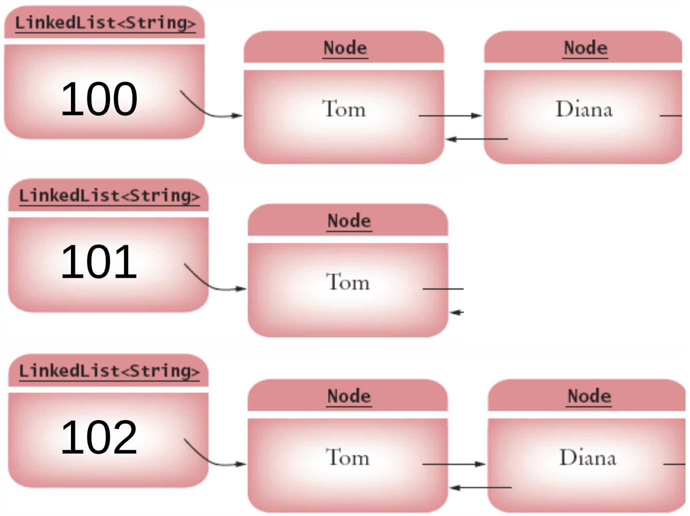
- `HashSet`: Stores data in a Hash Table
- `TreeSet`: Stores data in a Binary Tree
- Both implementations arrange the set elements so that finding, adding, and removing elements is efficient



Set implementations arrange the elements so that they can locate them quickly

Hash Table Concept

- Set elements are grouped into smaller collections of elements that share the same characteristic
  - It is usually based on the result of a mathematical calculation on the contents that results in an integer value
  - In order to be stored in a hash table, elements must have a method to compute their integer values



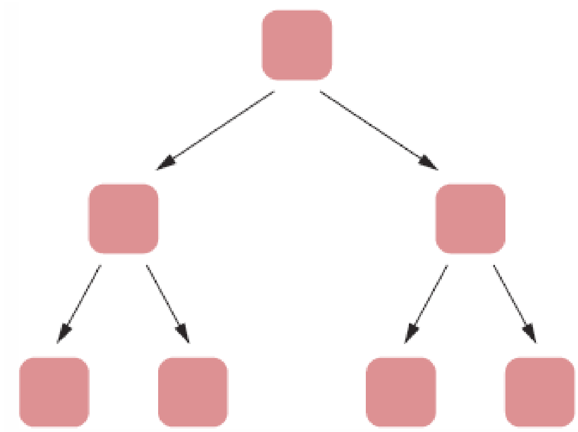
hashCode

- The method is called `hashCode`
  - If multiple elements have the same hash code, they are stored in a Linked list
- The elements must also have an `equals` method for checking whether an element equals another like:
  - String, Integer, Point, Rectangle, Color, and all collection classes

```
Set<String> names = new
   HashSet<String>();
```

Tree Concept

- Set elements are kept in sorted order
  - Nodes are not arranged in a linear sequence but in a tree shape



  - In order to use a `TreeSet`, it must be possible to compare the elements and determine which one is "larger"

TreeSet

- Use `TreeSet` for classes that implement the `Comparable` interface
  - `String` and `Integer`, for example
  - The nodes are arranged in a ‘tree’ fashion so that each ‘parent’ node has up to two child nodes.
    - The node to the left always has a ‘smaller’ value
    - The node to the right always has a ‘larger’ value

```
Set<String> names = new
   TreeSet<String>();
```

Iterators and Sets

- Iterators are also used when processing sets
  - `hasNext` returns true if there is a next element
  - `next` returns a reference to the value of the next element
  - `add` via the iterator is not supported for TreeSet and HashSet

```
Iterator<String> iter =
   names.iterator();
while (iter.hasNext())
{
  String name = iter.next();
  // Do something with name
}
```

```
for (String name : names)
{
  // Do something with name
}
```

- Note that the elements are not visited in the order in which you inserted them.
- They are visited in the order in which the set keeps them:
  - Seemingly random order for a HashSet
  - Sorted order for a TreeSet

Working With Sets (1)

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.

Working With Sets (2)

Table 4 Working with Sets

`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.
`for (String name : names) { . . . }`	Use this loop to visit all elements of a set.
`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

SpellCheck.java (1)

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
import java.io.File;
import java.io.FileNotFoundException;

/**
   This program checks which words in a file are not present in a dictionary.
*/
public class SpellCheck
{
   public static void main(String[] args)
      throws FileNotFoundException
   {
      // Read the dictionary and the document

      Set<String> dictionaryWords = readWords("words");
      Set<String> documentWords = readWords("alice30.txt");

      // Print all words that are in the document but not the dictionary

      for (String word : documentWords)
      {
         if (!dictionaryWords.contains(word))
         {
            System.out.println(word);
         }
      }
```

SpellCheck.java (2)

```java
    }

    /**
       Reads all words from a file.
       @param filename the name of the file
       @return a set with all lowercased words in the file. Here, a
       word is a sequence of upper- and lowercase letters.
    */
    public static Set<String> readWords(String filename)
       throws FileNotFoundException
    {
       Set<String> words = new HashSet<String>();
       Scanner in = new Scanner(new File(filename));
       // Use any characters other than a-z or A-Z as delimiters
       in.useDelimiter("[^a-zA-Z]+");
       while (in.hasNext())
       {
          words.add(in.next().toLowerCase());
       }
       return words;
    }
 }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

Programming Tip

- Use Interface References to Manipulate Data Structures
  - It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`.

```
Set<String> words = new
   HashSet<String>();
```

  - This way, you have to change only one line if you decide to use a `TreeSet` instead.

Programming Tip (continued)

- Unfortunately the same is not true of the `ArrayList`, `LinkedList` and `List` classes
    - The `get` and `set` methods for random access are very inefficient
- Also, if a method can operate on arbitrary collections, use the `Collection` interface type for the parameter:

```
public static void removeLongWords(Collection<String>
   words)
```

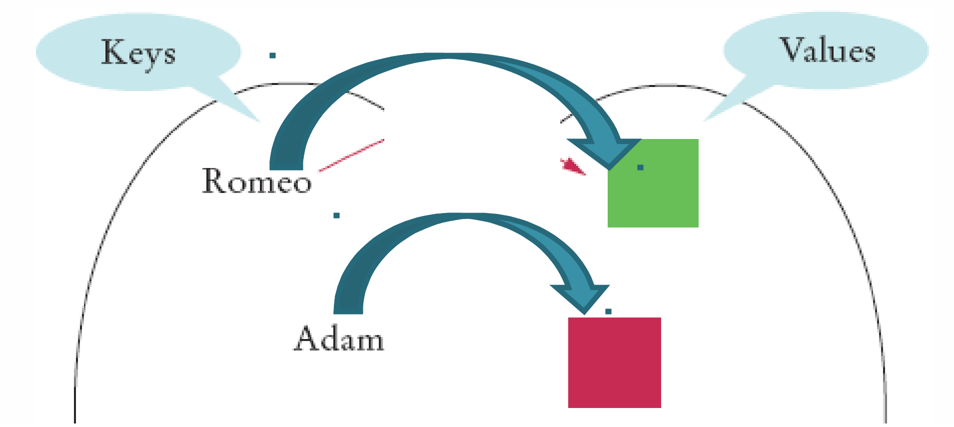
Maps

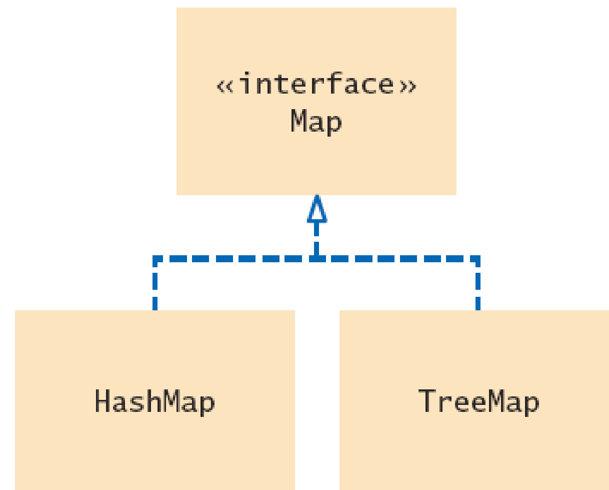
- A map allows you to associate elements from a key set with elements from a value collection.
  - The `HashMap` and `TreeMap` classes both implement the `Map` interface.
  - Use a map to look up objects by using a key.

Maps

Key Value Key Value

```
Map<String, Color> favoriteColors = new HashMap<String,
    Color>();
```





Working with Maps (Table 5)

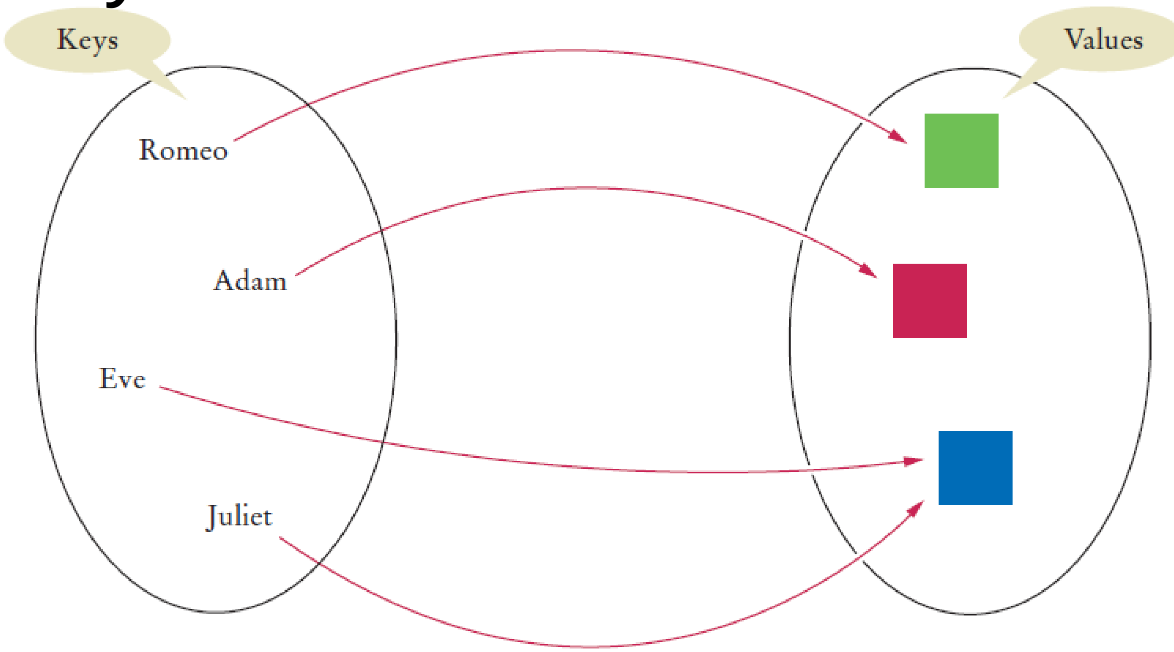
`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet())`
`{`
`   Integer value = scores.get(key);`
`   . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Key Value Pairs in Maps

- Each key is associated with a value




```
Map<String, Color> favoriteColors = new HashMap<String,
   Color>();
favoriteColors.put("Juliet", Color.RED);
favoriteColors.put(“Romeo", Color.GREEN);
Color julietsFavoriteColor = favoriteColors.get("Juliet");
favoriteColors.remove("Juliet");
```

Iterating through Maps

- To iterate through the map, use a `keySet` to get the list of keys:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
  Color value = m.get(key);
  System.out.println(key + "->" + value);
}
```

To find all values in a map, iterate through the key set and find the values that correspond to the keys.

MapDemo.java

```java
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
   This program demonstrates a map that maps names to colors.
*/
public class MapDemo
{
   public static void main(String[] args)
   {
      Map<String, Color> favoriteColors = new HashMap<String, Color>();
      favoriteColors.put("Juliet", Color.BLUE);
      favoriteColors.put("Romeo", Color.GREEN);
      favoriteColors.put("Adam", Color.RED);
      favoriteColors.put("Eve", Color.BLUE);

      // Print all keys and values in the map

      Set<String> keySet = favoriteColors.keySet();
      for (String key : keySet)
      {
         Color value = favoriteColors.get(key);
         System.out.println(key + " : " + value);
      }
   }
}
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

Steps to Choosing a Collection

1) Determine how you access values
   - Values are accessed by an integer position. Use an `ArrayList`
     - Go to Step 2, then stop
   - Values are accessed by a key that is not a part of the object
     - Use a `Map`.
   - It doesn’t matter. Values are always accessed “in bulk”, by traversing the collection and doing something with each value
2) Determine the element types or key/value types
   - For a `List` or `Set`, a single type
   - For a `Map`, the key type and the value type

Steps to Choosing a Collection

3) Determine whether element or key order matters

- Elements or keys must be sorted
  - Use a `TreeSet` or `TreeMap`. Go to Step 6
- Elements must be in the same order in which they were inserted
  - Your choice is now narrowed down to a `LinkedList` or an `ArrayList`
- It doesn’t matter
  - If you chose a map in Step 1, use a `HashMap` and go to Step 5

Steps to Choosing a Collection

4) For a collection, determine which operations must be fast

- Finding elements must be fast
  - Use a `HashSet` and go to Step 5
- Adding and removing elements at the beginning or the middle must be fast
  - Use a `LinkedList`
- You only insert at the end, or you collect so few elements that you aren’t concerned about speed
  - Use an `ArrayList`.

Steps to Choosing a Collection

5) For hash sets and maps, decide if you need to implement the `equals` and `hashCode` methods

- If your elements do not support them, you must implement them yourself.

6) If you use a tree, decide whether to supply a comparator

- If your element class does not provide it, implement the Comparable interface for your element class

Special Topic: Hash Functions

- Hashing can be used to find elements in a set data structure quickly, without making a linear search through all elements.
- A `hashCode` method computes and returns an integer value: the hash code.
  - Should be likely to yield different hash codes
  - Because hashing is so important, the Object class has a hashCode method that computes the hash code of any object x.

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

Computing Hash Codes

- To put objects of a given class into a `HashSet` or use the objects as keys in a `HashMap`, the class should override the default `hashCode` method.
- A good `hashCode` method should work such that different objects are likely to have different hash codes.
  - It should also be efficient
  - A simple example for a String might be:

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

Computing Hash Codes

- But `Strings` that are permutations of another (such as "eat" and "tea") would all have the same hash code
- Better:
  - From the Java Library!

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

Sample Strings and HashCodes

- The `String` class implements a good example of a `hashCode` method
- It is possible for two or more distinct objects to have the same hash code: This is called a **collision**
  - A `hashCode` function should minimizes collisions

Table 6 Sample Strings and Their Hash Codes

String	Hash Code
"eat"	100184
"tea"	114704
"Juliet"	–2065036585
"Ugh"	84982
"VII"	84982

Computing Object Hash Codes

- You should have a good `hashCode` method for your own objects to store them efficiently
- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables

```
public int hashCode()
{
   int h1 = name.hashCode();
   int h2 = new
    Double(area).hashCode();
```

- Then combine the hash codes using a prime-number hash multiplier:

```
   final int HASH_MULTIPLIER = 29;
   int h = HASH_MULTIPLIER * h1 + h2;
   return h;
}
```

`hashCode` and `equals` methods

- `hashCode` methods should be *compatible* with `equals` methods
  - If two objects are equal, their `hashCodes` should match
  - a `hashCode` method should use **all** instance variables
  - The `hashCode` method of the Object class uses the memory location of the object, not the contents

`hashCode` and `equals` methods

- Do not mix Object class `hashCode` or `equals` methods with your own:
  - Use an existing class such as String. Its `hashCode` and `equals` methods have already been implemented to work correctly.
  - Implement both `hashCode` and `equals`.
    - Derive the hash code from the instance variables that the `equals` method compares, so that equal objects have the same hash code
  - Implement neither `hashCode` nor `equals`. Then only identical objects are considered to be equal