# Java Programming

## Java Collections

# Review of Lecture 11

- Life Cycle of a thread
  - new
  - runnable
  - waiting, timed waiting
  - terminated
  - blocked
- Creating a thread
  - Create a class that implement Runnable interface
  - Implement run() method – thread's task.

- Creating and executing threads:

```
ExecutorService
   threadExecutor =
   Executors.newCachedThreadP
   ool();
// start threads and place in
   runnable state
threadExecutor.execute( task1
   ); // start task1
threadExecutor.execute( task2
   ); // start task2
```

- Thread synchronization
  - **built-in monitors** to implement synchronization
  - **synchronized statement**
  - **synchronized methods**

# Review of Lecture 11

- Can implement a shared using the synchronized keyword and Object methods **wait, notify and notifyAll.**
- Multithreading with GUI
  - Implement long running tasks in a **SwingWorker** thread.
  - Override **doInBackground** and done methods
- **Lock interface**
  - **lock and unlock methods**

```
public void run() {
      lock.lock();
      int y = cnt;
      cnt = y + 1;
      lock.unlock();
}
```

# Objectives

- Explain Java Collections Framework
- Use generic collection classes:
  - ArrayList
  - LinkedList
  - Queue
  - Stack
  - HashSet, and
  - HashMap

# Java Collections Framework

- Contain prepackaged data structures, interfaces, algorithms
  - Use generics
  - Use existing data structures
    - Example of code reuse
  - Provides reusable componentry

# Collections

- **Collection**
  - Data structure (object) that can hold references to other objects
- **Collections framework**
  - Interfaces declare operations for various collection types
  - Provide high-performance, high-quality implementations of common data structures
  - Enable software reuse
  - Enhanced with generics capabilities in J2SE 5.0
    - Compile-time type checking

# Java Collections Framework

- Some collection framework interfaces

| Interface | Description |
|---|---|
| `Collection` | The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived. |
| `Set` | A collection that does not contain duplicates. |
| `List` | An ordered collection that can contain duplicate elements. |
| `Map` | Associates keys to values and cannot contain duplicate keys. |
| `Queue` | Typically a first-in, first-out collection that models a waiting line; other orders can be specified. |

# Interface Collection and Class Collections

- Interface Collection contains bulk operations for **adding**, **clearing** and **comparing** objects in a collection.
- A Collection **can be converted to an array**.
- Interface Collection provides a method that returns an Iterator object, which allows a program to **walk through the collection** and remove elements from the collection during the iteration.
- Class Collections provides static methods that search, sort and perform other operations on collections.

# Java Collections Framework

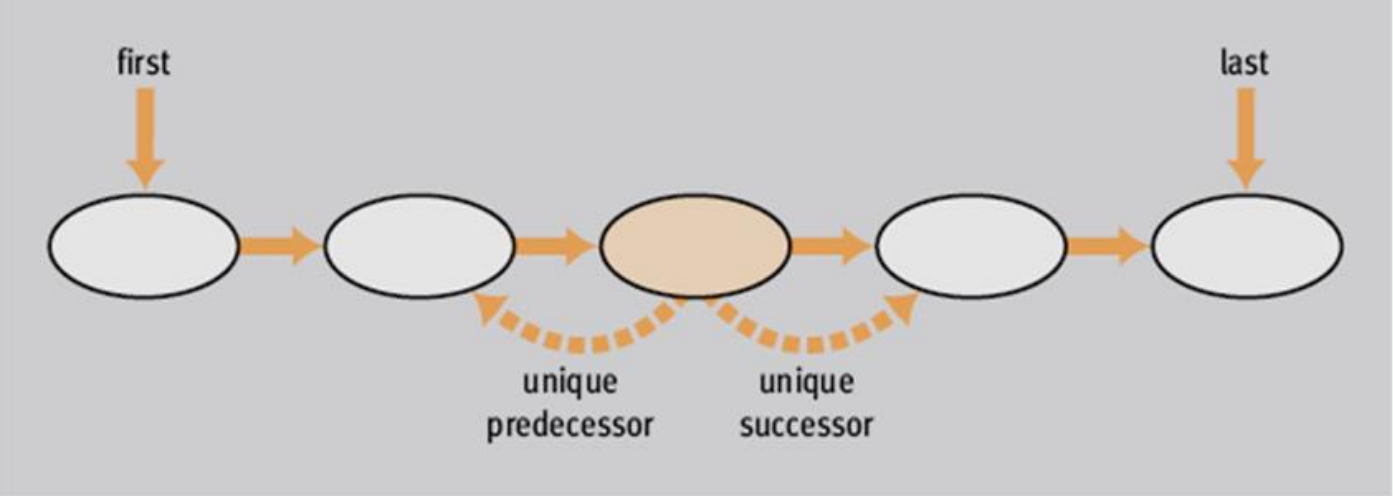
- Class `Arrays`
  - Provides `static` methods for manipulating arrays
  - Provides "high-level" methods
    - Method **`binarySearch`** for searching **sorted** arrays
    - Method **`equals`** for comparing arrays
    - Method **`fill`** for placing values into arrays
    - Method **`sort`** for sorting arrays

# Java Collections Framework

- Interface `Collection`
  - **Root** interface in the collection hierarchy
  - Interfaces **`Set`**, **`Queue`**, **`List`** extend interface **`Collection`**
    - **`Set`** – collection does not contain duplicates
    - **`Queue`** – collection represents a waiting line
    - **`List`** – ordered collection can contain duplicate elements
  - Contains *bulk operations*
    - Adding, clearing, comparing and retaining objects
  - Provide method to return an **Iterator** object
    - Walk through collection and remove elements from collection

# Java Collections Framework

- **List**
  - Ordered `Collection` that can contain duplicate elements



# Lists

- Implemented via interface `List`
  - **`ArrayList`** - `ArrayLists` behave like `Vectors` without synchronization and therefore execute faster than `Vectors` because `ArrayLists` do not have the overhead of thread synchronization.
  - **`LinkedList`** - `LinkedLists` can be used to create **stacks**, **queues**, **trees** and deques (double-ended queues, pronounced "**decks**").
  - The collections framework provides implementations of some of these data structures.
  - **`Vector`**

# ArrayList and Iterator

- List method add adds an item to the end of a list.
- List method size returns the number of elements.
- List method get retrieves an individual element's value from the specified index.
- Collection method iterator gets an Iterator for a Collection.
- Iterator- method hasNext determines whether there are more elements to iterate through.
  - Returns true if another element exists and false otherwise.
- Iterator method next obtains a reference to the next element.
- Collection method contains determine whether a Collection contains a specified element.
- Iterator method remove removes the current element from a Collection.

# Generic ArrayList

- **Raw type** ArrayList – no type of elements specified:
- ArrayList myList = new ArrayList()
  - Does not check the errors at compile time.

- **Generic** ArrayList – specifies the type of elements:

ArrayList<**String**> myList = new ArrayList<**String**>();

  - Checks the errors at compile time – you cannot add a non string element

# CollectionTest example

```
// add elements in colors array to list
String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
List<String> list = new ArrayList<String>();

for (String color : colors)
      list.add(color); // adds color to end of list

// add elements in removeColors array to removeList
String[] removeColors = {"RED", "WHITE", "BLUE"};
List<String> removeList = new ArrayList<String>();
for (String color : removeColors)
     removeList.add(color);
```

# ArrayList and Iterator

- Type Inference with the <> Notation
- The previous example specifies the type stored in the ArrayList (that is, String) on the left and right sides of the initialization statements.
- Java SE 7 introduced type inferencing with the <> notation—known as the **diamond notation**—in statements that declare and create generic type variables and objects. For example, line 14 can be written as:

```
List<String> list = new ArrayList<>();
```

- Java uses the type in angle brackets on the left of the declaration (that is, String) as the type stored in the ArrayList created on the right side of the declaration.

# LinkedList

- List method addAll appends all elements of a collection to the end of a List.
- List method listIterator gets A List's bidirectional iterator.
- String method toUpperCase gets an uppercase version of a String.
- List-Iterator method set replaces the current element to which the iterator refers with the specified object.
- String method toLowerCase returns a lowercase version of a String.
- List method subList obtaina a portion of a List.
- This is a so-called range-view method, which enables the program to view a portion of the list.

# LinkedList

- List method clear remove the elements of a List.
- List method size returns the number of items in the List.
- ListIterator method  hasPrevious determines whether there are more elements while traversing the list backward.
- ListIterator method previous gets the previous element from the list.

# LinkedList

- Class Arrays provides static method asList to view an array as a List collection.
- A List view allows you to manipulate the array as if it were a list.
- This is useful for adding the elements in an array to a collection and for sorting array elements.
- Any modifications made through the List view change the array, and any modifications made to the array change the List view.
- The only operation permitted on the view returned by asList is set, which **changes the value of the view and the backing array**.
- Any other attempts to change the view result in an UnsupportedOperationException.
- List method toArray gets an array from a List collection

# ListTest example

```
// add colors elements to list1
    String[] colors =
      {"black", "yellow", "green", "blue", "violet", "silver"};
    List<String> list1 = new LinkedList<>();

    for (String color : colors)
      list1.add(color);

    // add colors2 elements to list2
    String[] colors2 =
      {"gold", "white", "brown", "blue", "gray", "silver"};
    List<String> list2 = new LinkedList<>();

    for (String color : colors2)
      list2.add(color);
    list1.addAll(list2); // concatenate lists
```

# LinkedList

- LinkedList method addLast adds an element to the end of a List.
- LinkedList method add also adds an element to the end of a List.
- LinkedList method addFirst adds an element to the beginning of a List.

# Collections Algorithms

- Collections framework provides set of algorithms
  - Implemented as static methods
    - List algorithms
      - sort
      - binarySearch
      - reverse
      - shuffle
      - fill
      - Copy
    - Collection algorithms
      - min
      - max
      - addAll
      - frequency
      - disjoint

# Collections Algorithms

| Algorithm | Description |
|---|---|
| `sort` | Sorts the elements of a `List`. |
| `binarySearch` | Locates an object in a `List`. |
| `reverse` | Reverses the elements of a `List`. |
| `shuffle` | Randomly orders a `List`'s elements. |
| `fill` | Sets every `List` element to refer to a specified object. |
| `Copy` | Copies references from one `List` into another. |
| `min` | Returns the smallest element in a `Collection`. |
| `max` | Returns the largest element in a `Collection`. |
| `addAll` | Appends all elements in an array to a collection. |
| `frequency` | Calculates how many elements in the collection are equal to the specified element. |
| `disjoint` | Determines whether two collections have no elements in common. |

# Collections Algorithms - sort

- Method sort - sorts `List` elements
  - Order is determined by natural order of elements' type
  - `List` elements must implement the `Comparable` interface
  - Or, pass a `Comparator` to method `sort`
- Sorting in ascending order
  - Collections method **`sort`**
- Sorting in descending order
  - Collections static method `reverseOrder`
- Sorting with a `Comparator`
  - Create a custom `Comparator` class

# Collections Algorithms

- `shuffle`
  - Randomly orders `List` elements
- `reverse`
  - Reverses the order of `List` elements
- `fill`
  - Populates `List` elements with values
- `copy`
  - Creates copy of a `List`
- `max`
  - Returns largest element in `List`
- `min`
  - Returns smallest element in `List`

# Algorithms1 example

```
// create and display a List< Character >
    Character[] letters = {'P', 'C', 'M'};
    List<Character> list = Arrays.asList(letters); // get List
    System.out.println("list contains: ");
    output(list);

    // reverse and display the List<Character>
    Collections.reverse(list); // reverse order the elements
    System.out.printf("%nAfter calling reverse, list contains:%n");
    output(list);
```

# Method binarySearch

- static Collections method binarySearch locates an object in a List.
- If the object is found, its index is returned.
- If the object is not found, binarySearch returns a negative value.
- Method binarySearch determines this negative value by first calculating the insertion point and making its sign negative.
- Then, binarySearch subtracts 1 from the insertion point to obtain the return value, which guarantees that method binarySearch returns positive numbers (>= 0) if and only if the object is found.

# BinarySearchTest example

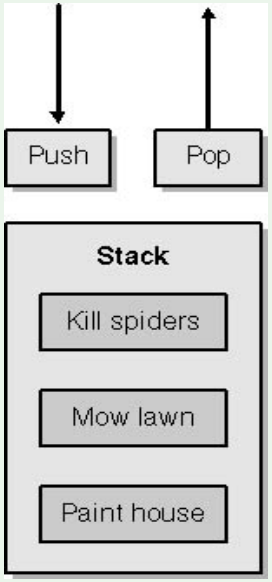
```
// perform search and display result
   private static void printSearchResults(
      List<String> list, String key)
   {
      int result = 0;
      System.out.printf("%nSearching for: %s%n", key);
      result = Collections.binarySearch(list, key);
      if (result >= 0)
         System.out.printf("Found at index %d%n", result);
      else
         System.out.printf("Not Found (%d)%n",result);
   }
```

# Collections Algorithms

- `addAll`
  - Insert all elements of an array into a collection
- `frequency`
  - Calculate the number of times a specific element appear in the collection
- `Disjoint`
  - Determine whether two collections have **elements in common**

# Stacks

- Last-in, first-out (LIFO) data structure
  - Method *push* **adds a new node** to the top of the stack
  - Method *pop* **removes a node from the top of the stack** and returns the data from the popped node
- Program execution stack
  - Holds the return addresses of calling methods
  - Also contains the local variables for called methods
  - Used by the compiler to evaluate arithmetic expressions

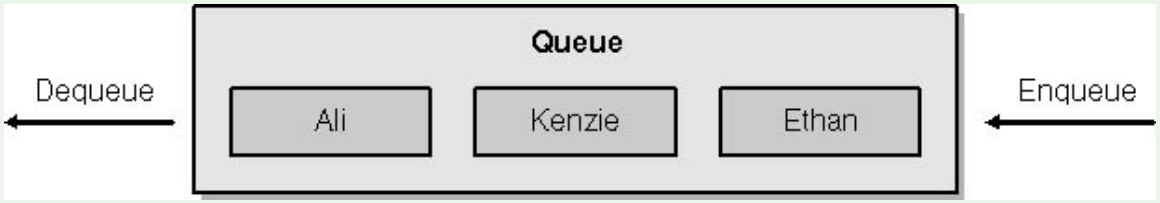


# Stack class – StackTest example

- Is in the Java utilities package (**java.util**) extends class Vector to implement a stack data structure.
- Stack method push adds a Number object to the top of the stack.
  - Any integer literal that has the suffix L is a long value.
  - An integer literal without a suffix is an int value.
  - Any floating-point literal that has the suffix F is a float value.
  - A floating-point literal without a suffix is a double value.
- Stack method pop removes the top element of the stack.
- If there are no elements in the Stack, method pop throws an EmptyStackException, which terminates the loop.
- Method peek returns the top element of the stack without popping the element off the stack.
- Method isEmpty determines whether the stack is empty.

# Queue

- Similar to a checkout line in a supermarket
  - First-in, first-out (FIFO) data structure
    - **Enqueue** inserts nodes at the tail (or end)
    - **Dequeue** removes nodes from the head (or front)
- Used to support print spooling
    - A spooler program manages the queue of printing jobs

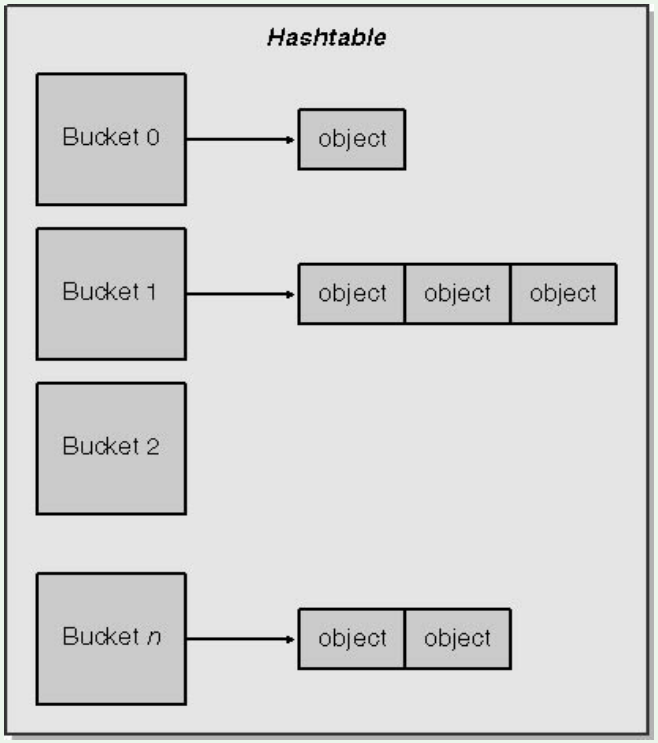


# Class PriorityQueue and Interface Queue – PriorityQueueTest example

- Interface Queue extends interface Collection and provides additional operations for inserting, removing and inspecting elements in a queue.
- PriorityQueue orders elements by their natural ordering.
- Elements are **inserted in priority order** such that the **highest-priority element** (i.e., the largest value) will be the first element removed from the PriorityQueue.
- Common PriorityQueue operations are:
  - offer to insert an element at the appropriate location based on priority order
  - poll to remove the highest-priority element of the priority queue
  - peek to get a reference to the highest-priority element of the priority queue
- clear to remove all elements in the priority queue  size to get the number of elements in the queue.

# What is a hashtable?

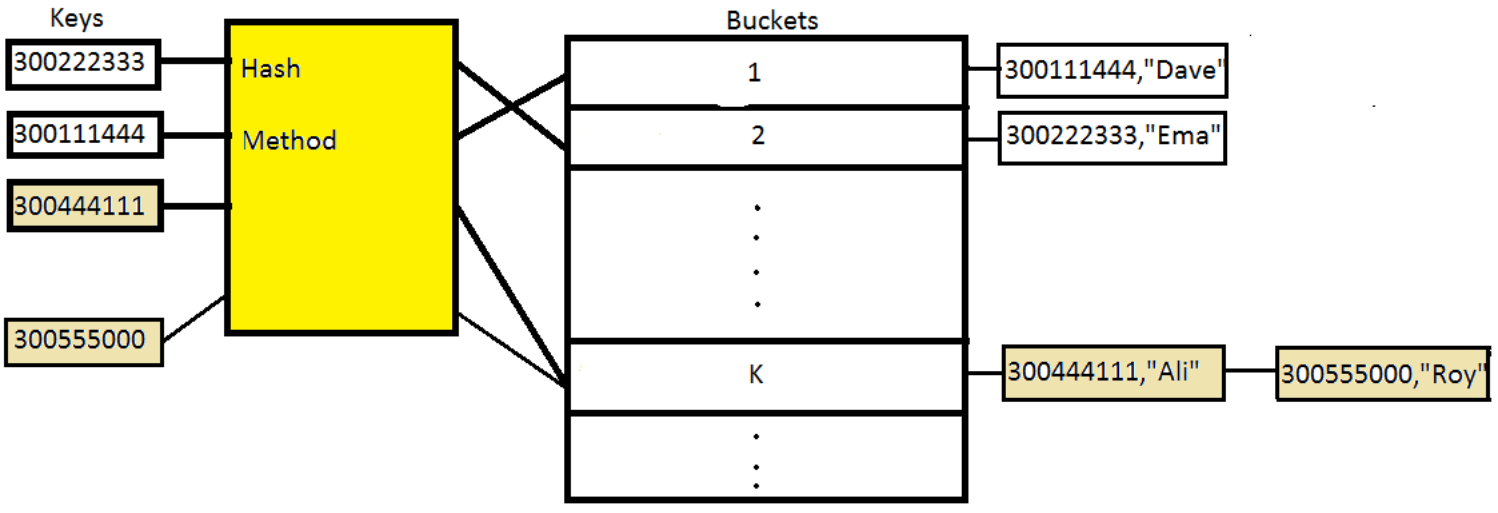
- A hash table is a collection that consists of **key-value** combinations, organized into '**buckets**' for fast searching.
- You can search a hash table either by the **keys**, or by the associated **values**
  - however, **searching by key will generally be faster**



# What is a hashtable?

- When one of these **key-value pairs** is added to the hash table, the hash table assigns to it a **hash code**, a number **that identifies the key**
  - each key you add gets a hash code
  - **hash codes are then placed in 'buckets'** to help organize the entries in the table.
  - This can help later when you try to find something in the table

# What is a hash table?




- For example, if the hash code equals one, the Hashtable places the value into Bucket 1, and so on.
- Multiple keys may map to the same bucket, something known as a *collision.*
- A linked list is used to maintain key/values in the same bucket.

# Sets – SetTest example

- A Set is an unordered Collection of unique elements (i.e., no duplicates).
- The collections framework contains several Set implementations, including HashSet and TreeSet.
- HashSet stores its elements in a hash table, and TreeSet stores its elements in a tree.

# Sets

- A set is a collection of elements without duplicates. For example, the following data describe the same set:

A={23, 43, 56, -100, 12, 7}
B={-100, 56, 7, 43, 23, 12 }

- A and B are the same, although the order of elements is different.

# Sets

- The Java collections library supplies a HashSet class that implements a set based on a hash table.
- You add elements with the add method.
  - The add method of a set first tries to find the object to be added, and adds it only if it is not yet present.

```
HashSet words = new HashSet();
words.add("Java");
```

- Use a HashSet if ordering of the elements in the collection is not important.

# Sets – SortedSetTest example

- The collections framework also includes the SortedSet interface (which extends Set) for sets that maintain their elements in sorted order.
- Class TreeSet implements SortedSet.
- TreeSet method headSet gets a subset of the TreeSet in which every element is less than the specified value.
- TreeSet method tailSet gets a subset in which each element is greater than or equal to the specified value.
- SortedSet methods first and last get the smallest and largest elements of the set, respectively.

# Maps

- Maps associate keys to values.
- The keys in a Map must be unique, but the associated values need not be.
- If a Map contains both unique keys and unique values, it is said to implement a one-to-one mapping.
- If only the keys are unique, the Map is said to implement a many-to-one mapping—many keys can map to one value.
- Three of the several classes that implement interface Map are Hashtable, HashMap and TreeMap.
- Hashtables and HashMaps store elements in hash tables, and TreeMaps store elements in trees.

# Maps

- Map method containsKey determines whether a key is in a map.
- Map method put creates a new entry or replaces an existing entry's value.
- Method put returns the key's prior associated value, or null if the key was not in the map.
- Map method get obtain the specified key's associated value in the map.
- HashMap method keySet returns a set of the keys.
- Map method size returns the number of key/value pairs in the Map.
- Map method isEmpty returns a boolean indicating whether the Map is empty.

# Maps – WordTypeCount example

- Interface SortedMap extends Map and maintains its keys in sorted order—either the elements' natural order or an order specified by a Comparator.
- Class TreeMap implements SortedMap.
- Hashing is a high-speed scheme for **converting keys into unique array indices**.
- A hash table's load factor affects the performance of hashing schemes.
- The **load factor** is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table.
- The closer this ratio gets to 1.0, the greater the chance of collisions.

# Class HashMap

- Here is how you set up a hash map for storing employees.

```
Map staff = new HashMap();
// create an employee object
// and put it in a HashMap
harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
. . .
```

- Whenever you add an object to a map, you must supply a key as well.
  - In this case, the key is a string, and the corresponding value is an Employee object

# Class HashMap

- To retrieve an object, you must use the key:

```
String s = "987-98-9996";
e = staff.get(s); // gets harry
```

- If no information is stored in the map with the particular key specified, then get returns null.

# Class HashMap

**Iterating through all entries in the table:**

- **Create an *iterator* object for the set of keys**
- **Use next method to get the next key**
- **Use get method to read the value associated with the key**

```
Set keySet = staff.keySet(); //get the set of keys
Iterator iter = keySet.iterator(); //iterator object
 while(iter.hasNext())
 {
        String key = (String)iter.next(); //get the next key
        Employee emp = (Employee)staff.get(key);
        System.out.println("key=" + key + ", value=" + emp.toString());

 }
```

# Class HashMap

```
/* get Collection of values contained in HashMap using
  Collection values() method of HashMap class    */
 Collection c = staff.values();
 //obtain an Iterator for Collection
 Iterator itr = c.iterator();
 //iterate through HashMap values iterator
 while(itr.hasNext())
  System.out.println(itr.next());
```

# Synchronized Collections

- Synchronization wrappers are used for collections that might be accessed by multiple threads.
- A wrapper object receives method calls, adds thread synchronization and delegates the calls to the wrapped collection object.
- The Collections API provides a set of static methods for wrapping collections as synchronized versions.
- Method headers for the synchronization wrappers are listed in Fig. 16.20 on next slide.

# Synchronized Collections

public static method headers

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
<T> List<T> synchronizedList(List<T> aList)
<T> Set<T> synchronizedSet(Set<T> s)
<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
<K, V> Map<K, V> synchronizedMap(Map<K, V> m)
<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)
```

Fig. 16.20 | Synchronization wrapper methods.

# References

- Textbook
- Java Documentation
  - https://docs.oracle.com/javase/tutorial/extra/generics/index.html
  - https://docs.oracle.com/javase/tutorial/collections/intro/index.html