CHAPTER 12

Thread, Network and Collections

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of Computer Science and Information Engineering
National Cheng Kung University

Thread

Multithreading

- In Java, programs can have multiple threads
  - A *thread* is a separate computation process
- Threads are often thought of as computations that run in parallel
  - Although they usually do not really execute in parallel
  - Instead, the computer switches resources between threads so that each one does a little bit of computing in turn
- Modern operating systems allow more than one program to run at the same time
  - An operating system uses threads to do this

Thread.sleep

- `Thread.sleep` is a static method in the class `Thread` that pauses the thread that includes the invocation
  - It pauses for the number of milliseconds given as an argument
  - Note that it may be invoked in an ordinary program to insert a pause in the single thread of that program
- It may throw a checked exception, `InterruptedException`, which must be caught or declared
  - Both the `Thread` and `InterruptedException` classes are in the package `java.lang`

Lab

```java
public class SleepTest {

  public static void main(String[] args) {

    System.out.println("Sleep for 3 seconds...");

    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Waked up");

  }

}
```

The Class Thread

- In Java, a thread is an object of the class **`Thread`**
- Usually, a derived class of **`Thread`** is used to program a thread
  - The methods **`run`** and **`start`** are inherited from **`Thread`**
  - The derived class overrides the method **`run`** to program the thread
  - The method **`start`** initiates the thread processing and invokes the **`run`** method

Lab

```
public class MyThread extends Thread{

        private String name;

        public MyThread(String name){
                this.name = name;
        }

        public void run(){

            while(true){
                System.out.println("Hello! I am " + name);

                try {
                        this.sleep((long)(Math.random()*2000));
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
            }
        }
}
```

Lab

```java
public class ThreadMain {
   public static void main(String[] args) {

      MyThread thread1 = new MyThread("Jimmy");
      thread1.start();
      MyThread thread2 = new MyThread("Lorenz");
      thread2.start();
      MyThread thread3 = new MyThread("Satriani");
      thread3.start();

   }
}
```

The `Runnable` Interface

- Another way to create a thread is to have a class implement the `Runnable` interface
  - The `Runnable` interface has one method heading:

    ```
    public void run();
    ```
- A class that implements `Runnable` must still be run from an instance of `Thread`
  - This is usually done by passing the `Runnable` object as an argument to the thread constructor

The **Runnable** Interface: Suggested Implementation Outline

```
public class ClassToRun extends SomeClass
    implements Runnable
{ . . .
    public void run()
    {
        // Fill this as if ClassToRun
        // were derived from Thread
    }
    . . .
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.run();
    }
    . . .
}
```

Lab

```java
public class Task implements Runnable{

  public void run(){
    while(true){
      System.out.println("This is a task");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

}
```

Lab

```java
public class TaskMain {

  public static void main(String[] args) {

    Thread thread = new Thread(new Task());
    thread.start();
  }

}
```

Race Conditions

- When multiple threads change a shared variable it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value.
- This is called a race condition because the final value depends on the sequence in which the threads access the shared value.
- We will use the Counter class to demonstrate a race condition.

Counter Class

Display 19.4 The `Counter` Class

```
public class Counter
{
  private int counter;
  public Counter()
  {
         counter = 0;
  }
  public int value()
  {
         return counter;
  }
  public void increment()
  {
         int local;
         local = counter;
         local++;
         counter = local;
  }
}
```

Race Condition Example

1. Create a single instance of the Counter class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the Counter class.
3. Each thread runs and invokes the *increment()* method.
4. Wait for each thread to finish and then output the value of the counter. If there were no race conditions then its value should be 30,000. If there were race conditions then the value will be less than 30,000.

Race Condition Test Class (1 of 3)

Display 19.5 The `RaceConditionTest` Class

```
public class RaceConditionTest extends Thread
{
  private Counter countObject;

  public RaceConditionTest(Counter ctr)
  {
       countObject = ctr;
  }
```

Stores a reference to a single `Counter` object.

Race Condition Test Class (2 of 3)

```java
    public void run()
    {
      countObject.increment();
    }

    public static void main(String[] args)
    {
      int i;
      Counter masterCounter = new Counter();
      RaceConditionTest[] threads = new RaceConditionTest[30000];

      System.out.println("The counter is " + masterCounter.value());
      for (i = 0; i < threads.length; i++)
      {
            threads[i] = new RaceConditionTest(masterCounter);
            threads[i].start();
      }
```

Invokes the code in Display 19.4 where the race condition occurs.

The single instance of the `Counter` object.

Array of 30,000 threads.

Give each thread a reference to the single `Counter` object and start each thread.

Race Condition Test Class (3 of 3)

```
        // Wait for the threads to finish
        for (i = 0; i < threads.length; i++)
        {
            try
            {
                threads[i].join();
            }
            catch (InterruptedException e)
            {
                System.out.println(e.getMessage());
            }
        }
        System.out.println("The counter is " + masterCounter.value());
    }
}
```

Waits for the thread to complete.

Sample Dialogue (output will vary)

```
The counter is 0
The counter is 29998
```

Lab

```java
public class RaceConditionTest {

	private static long count;

	public static void increase(){
		long local = count;
		local++;
		count=local;
	}

	public static void main(String[] args) {

		//starts all threads
		Racer[] racer = new Racer[30000];
		for(int i=0;i<30000;i++){
			racer[i] = new Racer();
			racer[i].start();
		}

		//waits for all theads to complete
		for(int i=0;i<30000;i++){
			try {
				racer[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		System.out.println("count="+count);
	}

}
```

Lab

```java
public class Racer extends Thread{

        public void run(){
                RaceConditionTest.increase();
        }
}
```

Thread Synchronization

- The solution is to make each thread wait so only one thread can run the code in increment() at a time.
- This section of code is called a **critical region** . Java allows you to add the keyword **synchronized** around a critical region to enforce that only one thread can run this code at a time.

Synchronized

- Two solutions:

```
public synchronized void increment()
{
  int local;
  local = counter;
  local++;
  counter = local;
}
```

```
public void increment()
{
  int local;
  synchronized (this)
  {
   local = counter;
   local++;
   counter = local;
  }
}
```

Lab

```java
public class RaceConditionTest {

	private static long count;

	public synchronized static void increase(){
		long local = count;
		local++;
		count=local;
	}

	public static void main(String[] args) {

		//starts all threads
		Racer[] racer = new Racer[30000];
		for(int i=0;i<30000;i++){
			racer[i] = new Racer();
			racer[i].start();
		}

		//waits for all theads to complete
		for(int i=0;i<30000;i++){
			try {
				racer[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		System.out.println("count="+count);
	}

}
```

Network

Networking with Stream Sockets

- Transmission Control Protocol – **TCP**
  - Most common network protocol on the Internet
  - Called a reliable protocol because it guarantees that data sent from the sender is received in the same order it is sent
- **Server**
  - Program waiting to receive input
- **Client**
  - Program that initiates a connection to the server

Sockets

- A socket describes one end of the connection between two programs over the network. It consists of:
  - An address that identifies the remote computer, e.g. IP Address
  - A port for the local and remote computer
    - Number between 0 and 65535
    - Identifies the program that should handle data received by the network
    - Only one program may bind to a port
    - Ports 0 to 1024 are reserved for the operating system

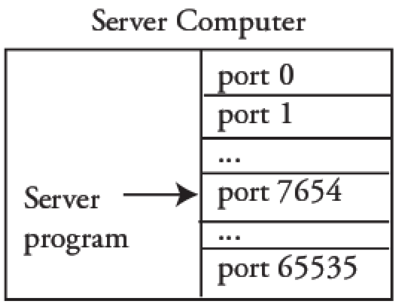
Sockets Programming

- Very similar to File I/O using a *FileOutputStream* but instead we substitute a *DataOutputStream*
- We can use *localhost* as the name of the local machine
- Socket and stream objects throw checked exceptions
  - We must catch them

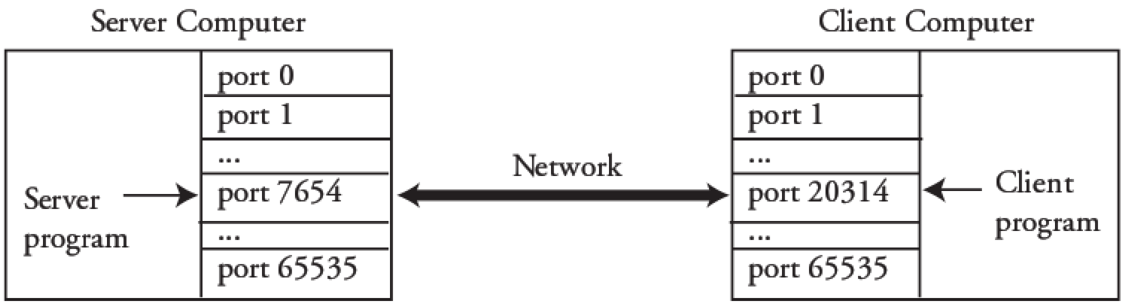
Client/Server Socket Example

Display 19.4 Client/Server Network Communication through Sockets

1. The server listens and waits for a connection on port 7654.



2. The client connects to the server on port 7654. It uses a local port that is assigned automatically, in this case, port 20314.



The server program can now communicate over a socket bound locally to port 7654 and remotely to the client's address at port 20314

The client program can now communicate over a socket bound locally to port 20314 and remotely to the server's address at port 7654

Lab (Server Side)

```java
package chapter12;
import java.net.*;
import java.io.*;

public class SocketServer{
	public static void main(String[] args){
		try{
			//Waiting for a connection on port 8000.
			ServerSocket serverSock = new ServerSocket(8000);
			Socket connectionSock = serverSock.accept();

			BufferedReader clientInput = new BufferedReader(new InputStreamReader(connectionSock.getInputStream()));
			DataOutputStream clientOutput = new DataOutputStream(connectionSock.getOutputStream());

			//Connection made, waiting for client to send their name.
			String clientText = clientInput.readLine();
			System.out.println("From Client: "+clientText);

			//reply message
			String replyText = "Hello! I am the server...\n" ;
			clientOutput.writeBytes(replyText);

			clientOutput.close();
			clientInput.close();
			connectionSock.close();
			serverSock.close();
		}
		catch (IOException e)
		{
		System.out.println(e.getMessage());
		}
	}
}
```

Lab (Client Side)

```
import java.net.*;
import java.io.*;

public class SocketClient{
      public static void main(String[] args) {
            try{

                  //Connecting to server on port 8000
                  Socket cSock = new Socket("127.0.0.1", 8000);

                  BufferedReader serverInput = new BufferedReader(new InputStreamReader(cSock.getInputStream()));
                  DataOutputStream serverOutput = new DataOutputStream(cSock.getOutputStream());

                  //Connection made, sending name.;
                  serverOutput.writeBytes("Hello! I am a client...\n");

                  //Waiting for reply.
                  String serverData = serverInput.readLine();
                  System.out.println("From Server: " + serverData);

                  serverOutput.close();
                  serverInput.close();
                  cSock.close();
            }
            catch (IOException e)
            {
                        System.out.println(e.getMessage());
            }
      }
}
```

Sockets and Threading

- The server waits, or blocks, at the *serverSock.accept()* call until a client connects.
- The client and server block at the *readLine()* calls if data is not available.
- This can cause an unresponsive network program and difficult to handle connections from multiple clients on the server end
- The typical solution is to employ threading

Threaded Server

- For the server, the *accept()* call is typically placed in a loop and a new thread created to handle each client connection:

```
while (true)
{
        Socket connectionSock = serverSock.accept( );
        ClientHandler handler = new ClientHandler(connectionSock);
        Thread theThread = new Thread(handler);
        theThread.start( );
}
```

Lab (Server Side)

```java
import java.io.*;
import java.net.*;
public class ChatRoom {
	public static void main(String[] args) {
		try {
			ServerSocket serverSock = new ServerSocket(8000);
			System.out.print("Server started...");

			while (true) {
				Socket cSock = serverSock.accept();
				Chat chat = new Chat(cSock);
				Thread chatthread = new Thread(chat);
				chatthread.start();
			}
		} catch (IOException e) { System.out.println("disconnected..."); }
	}
}
class Chat implements Runnable {
	private Socket socket;
	public Chat(Socket socket) {  this.socket = socket; }
	public void run() {
		try {
			BufferedReader clientInput = new BufferedReader(new InputStreamReader(socket.getInputStream()));
			while(true){
				String clientText = clientInput.readLine();
				System.out.println("From Client:  " + clientText);
				if(clientText.equals("bye")) break;
			}
			clientInput.close();
			socket.close();
		} catch (Exception e) { e.printStackTrace(); }
	}
}
```

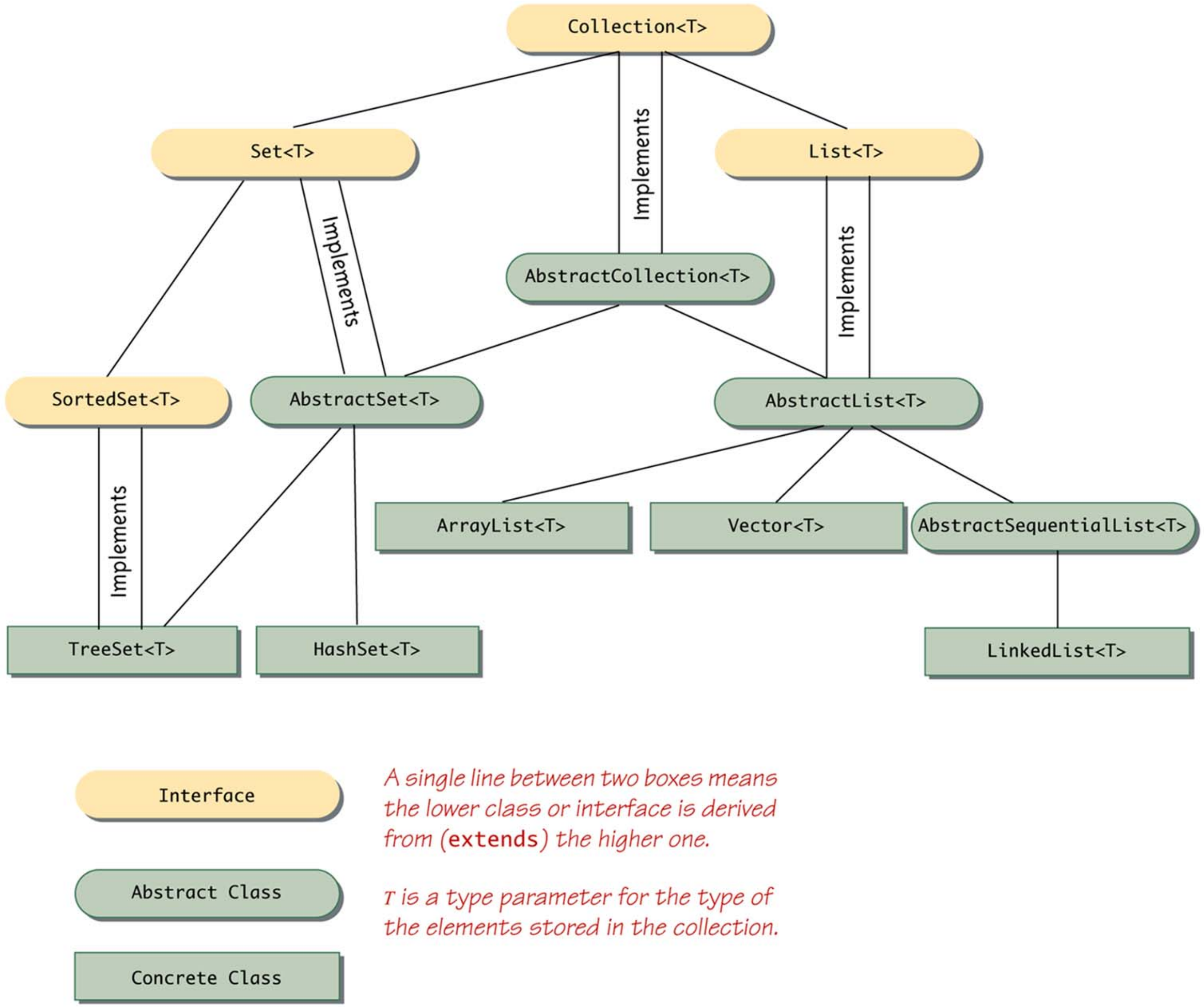
Lab (Client Side)

```java
import java.io.*;
import java.net.Socket;
import java.util.Scanner;
public class ChatClient {
	public static void main(String[] args) {
		try {
			// Connecting to server on port 8000
			Socket connectionSock = new Socket("127.0.0.1", 8000);

			DataOutputStream serverOutput = new DataOutputStream(connectionSock.getOutputStream());

			// Connection made, sending name.;
			while (true) {
				Scanner scanner = new Scanner(System.in);
				System.out.println("Type your message:");
				String msg = scanner.nextLine();

				if(!msg.equals("")) serverOutput.writeBytes(msg + "\n");
				if (msg.equals("bye"))break;
			}

			serverOutput.close();
			connectionSock.close();
		} catch (IOException e) {
				e.printStackTrace());
		}
	}
}
```

Collections and Maps

The Collection Landscape

Display 16.1 **The Collection Landscape**



The `Collection` Framework

- The `Collection<T>` interface describes the basic operations that all collection classes should implement
  - The method headings for these operations are shown on the next several slides
- Since an interface is a type, any method can be defined with a parameter of type `Collection<T>`
  - That parameter can be filled with an argument that is an object of any class in the collection framework

Method Headings in the **`Collection<T>`** Interface (Part 1 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

The `Collection<T>` interface is in the `java.util` package.
All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause.
All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Collection<T>` interface should have at least two constructors: a no-argument constructor that creates an empty `Collection<T>` object, and a constructor with one parameter of type `Collection<? extends T>` that creates a `Collection<T>` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

(continued)

Method Headings in the `Collection<T>` Interface (Part 2 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public boolean contains(Object target)
```

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

(continued)

Method Headings in the **`Collection<T>`** Interface (Part 3 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public boolean containsAll(Collection<?> collectionOfTargets)
```

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an element in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object.
Throws a `ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).
Throws a `NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionOfTargets` is `null`.

```
public boolean equals(Object other)
```

This is the `equals` of the collection, not the `equals` of the elements in the collection. Overrides the inherited method `equals`. Although there are no official constraints on `equals` for a collection, it should be defined as we have described in Chapter 7 and also to satisfy the intuitive notion of collections being equal.

(continued)

Method Headings in the `Collection<T>` Interface (Part 4 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public int size()
```

Returns the number of elements in the calling object. If the calling object contains more than `Inte–ger.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

```
Iterator<T> iterator()
```

Returns an iterator for the calling object. (Iterators are discussed in Section 16.2.)

```
public Object[] toArray()
```

Returns an array containing all of the elements in the calling object. If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The array returned should be a new array so that the calling object has no references to the returned array. (You might also want the elements in the array to be clones of the elements in the collection. However, this is apparently not required by the interface, since library classes, such as `Vector<T>`, return arrays that contain references to the elements in the collection.)

(continued)

Method Headings in the `Collection<T>` Interface (Part 5 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public <E> E[] toArray(E[] a)
```

Note that the type parameter E is not the same as T. So, E can be any reference type; it need not be the type T in `Collection<T>`. For example, E might be an ancestor type of T.

Returns an array containing all of the elements in the calling object. The argument *a* is used primarily to specify the type of the array returned. The exact details are as follows:

The type of the returned array is that of *a*. If the elements in the calling object fit in the array *a*, then *a* is used to hold the elements of the returned array; otherwise a new array is created with the same type as *a*.

If *a* has more elements than the calling object, the element in *a* immediately following the end of the copied elements is set to `null`.

If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order. (Iterators are discussed in Section 16.2.)

Throws an `ArrayStoreException` if the type of *a* is not an ancestor type of the type of every element in the calling object.

Throws a `NullPointerException` if *a* is `null`.

(continued)

Method Headings in the `Collection<T>` Interface (Part 6 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public int hashCode()
```

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is only here to make the definition of the `Collection<T>` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have the method throw an `UnsupportedOperationException`.

OPTIONAL METHODS

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if, for some reason, you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

(continued)

Method Headings in the `Collection<T>` Interface (Part 7 of 10)

Display 16.2 **Method Headings in the `Collection<T>` Interface**

```
public boolean add(T element)
```
(Optional)

Ensures that the calling object contains the specified `element`. Returns `true` if the calling object changed as a result of the call. Returns `false` if the calling object does not permit duplicates and already contains `element`; also returns `false` if the calling object does not change for any other reason.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the class of `element` prevents it from being added to the calling object.
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some other aspect of `element` prevents it from being added to the calling object.

(continued)

Method Headings in the **`Collection<T>`** Interface (Part 8 of 10)

Display 16.2 **Method Headings in the `Collection<T>` Interface**

```
public boolean addAll(Collection<? extends T> collectionToAdd) (Optional)
```

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise. If the calling object changes during this operation, its behavior is unspecified; in particular, its behavior is unspecified if `collectionToAdd` is the calling object.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

Throws a `ClassCastException` if the class of an element of `collectionToAdd` prevents it from being added to the calling object.

Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

Throws an `IllegalArgumentException` if some aspect of an element of `collectionToAdd` prevents it from being added to the calling object.

(continued)

Method Headings in the `Collection<T>` Interface (Part 9 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public boolean remove(Object element)` *(Optional)*

Removes a single instance of the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`public boolean removeAll(Collection<?> collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling collection (optional).
Throws a `NullPointerException` if `collectionToRemove` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionToRemove` is `null`.

(continued)

Method Headings in the **`Collection<T>`** Interface (Part 10 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public void clear()` *(Optional)*

Removes all the elements from the calling object.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`public boolean retainAll(Collection<?> saveElements)` *(Optional)*

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the types of one or more elements in `saveElements` are incompatible with the calling object (optional).
Throws a `NullPointerException` if `saveElements` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `saveElements` is `null`.

Concrete Collections Classes

- The concrete class **`HashSet<T>`** implements the **`Set<T>`** interface
  - The **`HashSet<T>`** class is implemented using a *hash table*
- The interface **`SortedSet<T>`** and the concrete class **`TreeSet<T>`** are designed for implementations of the **`Set<T>`** interface that provide for rapid retrieval of elements
  - The implementation of the class is similar to a binary tree, but with ways to do inserting that keep the tree balanced
- HashSet is much faster than TreeSet (constant-time versus log-time for most operations like add, remove and contains) but offers no ordering guarantees like TreeSet.

Lab

```java
import java.util.*;
public class SetTest {

	public static void main(String[] args) {
		TreeSet treeset = new TreeSet();
		treeset.add("c");
		treeset.add("b");
		treeset.add("a");

		Iterator itr1 = treeset.iterator();
		while(itr1.hasNext()){
				System.out.println(itr1.next());
		}

		HashSet hashset = new HashSet();
		hashset.add("c");
		hashset.add("b");
		hashset.add("a");

		Iterator itr2 = hashset.iterator();
		while(itr2.hasNext()){
				System.out.println(itr2.next());
		}
	}
}
```

Concrete Collections Classes

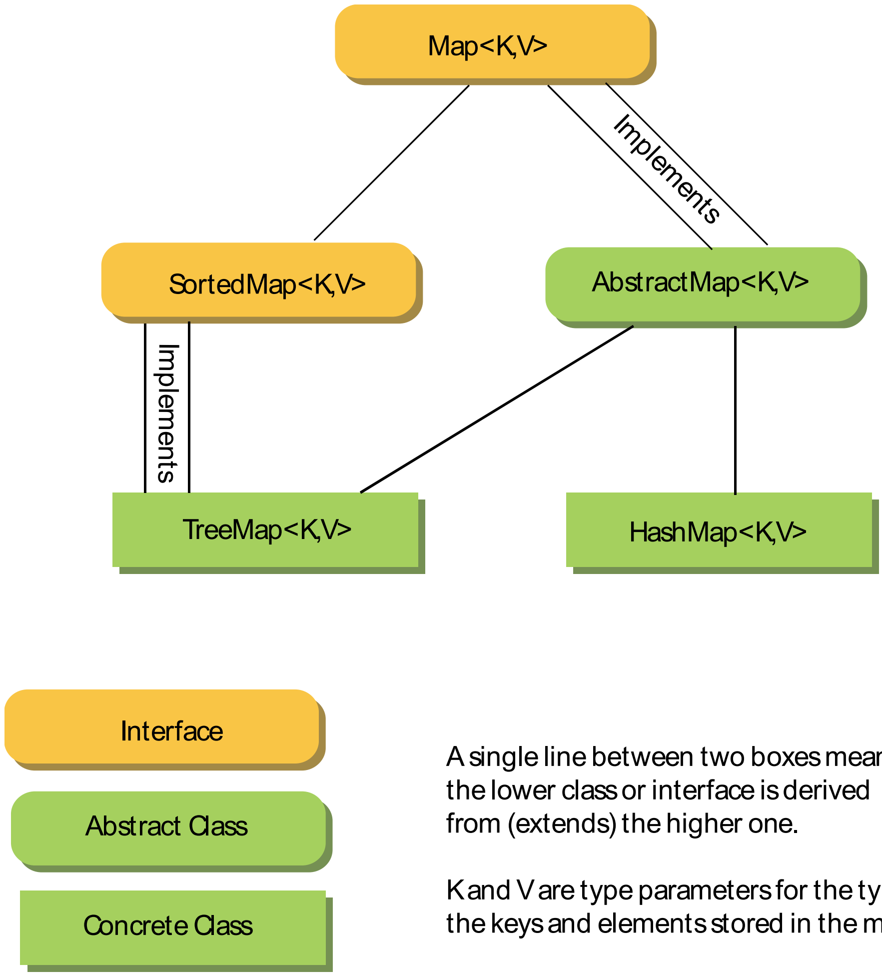
- The **`ArrayList<T>`** and **`Vector<T>`** classes implement the **`List<T>`** interface
  - ArrayList is non-synchronized which means multiple threads can work on ArrayList at the same time.
  - ArrayList gives better performance as it is non-synchronized.

- The concrete class **`LinkedList<T>`** is a concrete derived class of the abstract class **`AbstractSequentialList<T>`**
  - get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n)
  - LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Lab

```java
import java.util.ArrayList;

public class ListTest {

    public static void main(String[] args) {
        ArrayList alist = new ArrayList();
        alist.add("c");
        alist.add("b");
        alist.add("a");

        for(int i=0;i<alist.size();i++){
                System.out.println(alist.get(i));
        }
    }

}
```

The Map Landscape



A single line between two boxes means the lower class or interface is derived from (extends) the higher one.

K and V are type parameters for the type of the keys and elements stored in the map.

The Map<K,V> Interface (1 of 3)

Display 16.9 Method Headings in the Map<K,V> Interface

The `Map<K,V>` interface is in the `java.util` package.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Map<K,V>` interface should have at least two constructors: a no-argument constructor that creates an empty `Map<K,V>` object, and a constructor with one `Map<K,V>` parameter that creates a `Map<K,V>` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

METHODS

```
boolean isEmpty( )
```

Returns `true` if the calling object is empty; otherwise returns `false`.

```
public boolean containsValue(Object value)
```

Returns `true` if the calling object contains at least one or more keys that map to an instance of `value`.

```
public boolean containsKey(Object key)
```

Returns `true` if the calling object contains `key` as one of its keys.

The Map<K,V> Interface (2 of 3)

`public boolean equals(Object other)`

This is the **equals** of the map, not the **equals** of the elements in the map. Overrides the inherited method **equals**.

`public int size( )`

Returns the number of (key, value) mappings in the calling object.

`public int hashCode( )`

Returns the hash code value for the calling object.

`public Set<Map.Entry<K,V>> entrySet( )`

Returns a set *view* consisting of (key, value) mappings for all entries in the map. Changes to the map are reflected in the set and vice-versa.

`public Collection<V> values( )`

Returns a collection *view* consisting of all values in the map. Changes to the map are reflected in the collection and vice-versa.

`public V get(Object key)`

Returns the value to which the calling object maps **key**. If **key** is not in the map, then **null** is returned. Note that this does not always mean that the key is not in the map since it is possible to map a key to **null**. The **containsKey** method can be used to distinguish the two cases.

The Map<K,V> Interface (3 of 3)

OPTIONAL METHODS

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if, for some reason, you do not want to give the methods a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

`public V put(K key, V value)` *(Optional)*

Associates `key` to `value` in the map. If `key` was associated with an existing value then the old value is overwritten and returned. Otherwise `null` is returned.

`public void putAll(Map<? extends K,? extends V> mapToAdd)` *(Optional)*

Adds all mappings of `mapToAdd` into the calling object's map.

`public V remove(Object key)` *(Optional)*

Removes the mapping for the specified key. If the key is not found in the map then `null` is returned; otherwise the previous value for the key is returned.

Concrete Map Classes

- `HashMap<K,V>` Class
  - No guarantee as to the order of elements placed in the map.
- `TreeMap<K,V>` class
  - If you require order then you should use the TreeMap

Lab

```java
import java.util.*;

public class MapTest {

    public static void main(String[] args) {

        HashMap hashmap = new HashMap();
        hashmap.put("Joy", "175");
        hashmap.put("Sandy", "178");
        hashmap.put("Blue", "166");

        Iterator itr1 = hashmap.keySet().iterator();
        while(itr1.hasNext()){
            String key = (String)itr1.next();
            String value = (String)hashmap.get(key);
            System.out.println(key +":" + value);
        }
    }

}
```

Lab

```java
import java.util.*;

public class MapTest {

	public static void main(String[] args) {

		HashMap hashmap = new HashMap();
		hashmap.put("Joy", "175");
		hashmap.put("Sandy", "178");
		hashmap.put("Blue", "166");

		Iterator itr1 = hashmap.keySet().iterator();
		while(itr1.hasNext()){
			String key = (String)itr1.next();
			String value = (String)hashmap.get(key);
			System.out.println(key +":" + value);
		}

		TreeMap treemap = new TreeMap();
		treemap.put("Joy", "175");
		treemap.put("Sandy", "178");
		treemap.put("Blue", "166");
		Iterator itr2 = treemap.keySet().iterator();
		while(itr2.hasNext()){
			String key = (String)itr2.next();
			String value = (String)hashmap.get(key);
			System.out.println(key +":" + value);
		}
	}

}
```