aws PARTNER Premier Tier Services

Microsoft Solutions Partner

Google Cloud Partner

CloudKeeper

Java Collections & Multithreading

Proudly recognized by

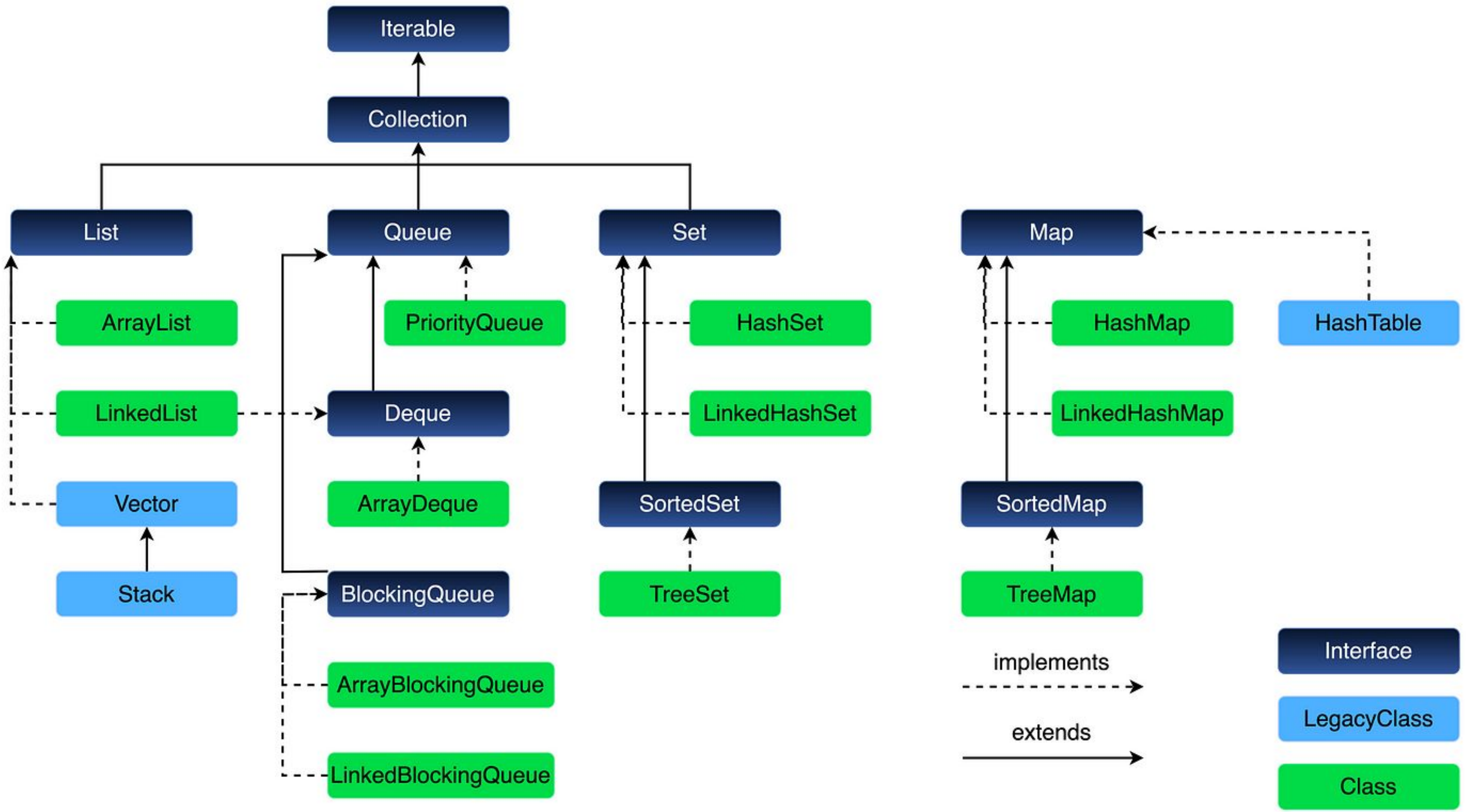
Agenda

1. Collections
    a. List Interface
    b. Set Interface
    c. Map Interface
2. Multi-threading
    a. Multi-threading Terminology
    b. Thread Life Cycle
    c. Creating Threads
    d. Sleep and Join
    e. Thread Pools

Collections

The Java platform includes a collections framework. A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

Collections



List Interface

- The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.

- A list may contain duplicate elements.

- Several of the list methods will throw an `UnsupportedOperationException` if the collection cannot be modified, and a `ClassCastException` is generated when one object is incompatible with another.

List Interface (cont.)

- Implementations of List interface:
    - **ArrayList**: Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.
    - **LinkedList**: Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).
    - **Vector**: The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
    - **Stack**: The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack.

List Interface (cont.)

ArrayList vs Vector

- **synchronization** – The first major difference between these two. Vector is synchronized and ArrayList isn't.
- **size growth** – Another difference between the two is the way they resize while reaching their capacity. The Vector doubles its size. In contrast, ArrayList increases only by half of its length
- **iteration** – And Vector can use Iterator and Enumeration to traverse over the elements. On the other hand, ArrayList can only use Iterator.
- **performance** – Largely due to synchronization, Vector operations are slower when compared to ArrayList
- **framework** – Also, ArrayList is a part of the Collections framework and was introduced in JDK 1.2. Meanwhile, Vector is present in the earlier versions of Java as a legacy class.

Set Interface

- A collection that contains no duplicate elements.
- More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.
- The Set interface places additional stipulations, beyond those inherited from the Collection interface, on the contracts of all constructors and on the contracts of the add, equals and hashCode methods.

Set Interface (cont.)

- Implementations of Set interface:
    - **HashSet:** This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.
    - **LinkedHashSet**: Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order).
    - **TreeSet**: A Set that further provides a total ordering on its elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering.

Map Interface

- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

- The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

Map Interface (cont.)

- Implementations of Map interface:
  - **HashMap:** This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.
  - **LinkedHashMap**: Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.
  - **TreeMap**: A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

Multi-threading

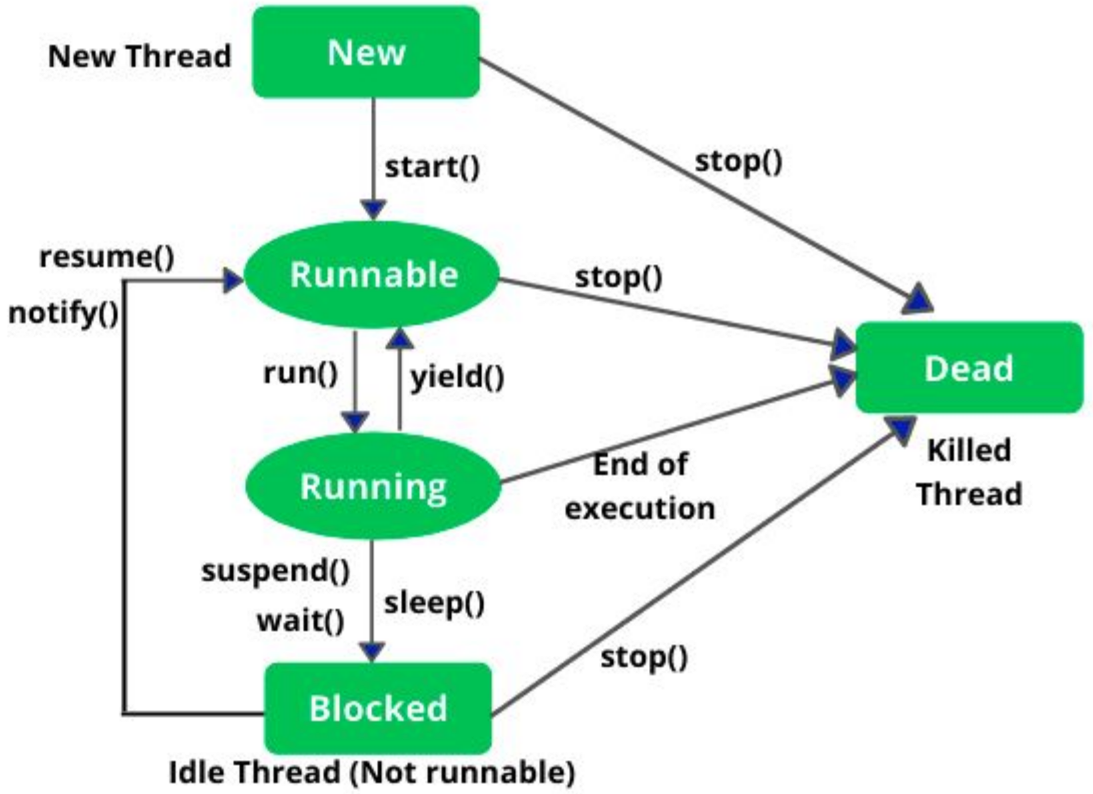
- A process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Multi-threading Terminologies

- **Thread**: Smallest unit of execution that can be scheduled by Operating System.
- **Process**: Group of associated threads that execute in the same, shared environment.
- **Task**: Single unit of work performed by thread. A Thread can complete multiple independent tasks but only one task at a time.
- **Multiprocessing**: It uses multiple CPUs to run many processes at a time. Execution of one process won't be affected by any other process in execution.
- **Thread Priority**: A Thread Priority is numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executed.
  - Thread.MIN_PRIORITY: Value is 1
  - Thread.NORM_PRIORITY: Value is 5. This is default priority.
  - Thread.MAX_PRIORITY: Value is 10.

Thread Life Cycle

CloudKeeper



Thread Life Cycle (cont.)

- **New**: When we create a new Thread object using new operator, thread state is New Thread. At this point, thread is not alive and it's a state internal to Java programming.
- **Runnable**: When we call start() function on Thread object, it's state is changed to Runnable.
- **Running**: When thread is executing, it's state is changed to Running.
- **Blocked/Waiting**: A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available.
- **Dead**: Once the thread finished executing, it's state is changed to Dead and it's considered to be not alive.

Creating Threads

- A thread can be created by extending **Thread** class in java.lang package.

```
public class ThreadDemo extends Thread{
    @Override
    public void run(){
        System.out.println("Running Thread Demo");
    }

    public static void main(String[] args) {
        new ThreadDemo().start();
    }
}
```

Creating Threads (cont.)

- Implementation of **Runnable** interface can be passed to thread class object.

```
public class RunnableDemo implements Runnable{
    @Override
    public void run() {
        System.out.println("Running Runnable Thread");
    }


    public static void main(String[] args) {
        new Thread(new RunnableDemo()).start();
    }
}
```

Creating Threads (cont.)

- Creating a Thread using anonymous inner class

```
public class SimpleThread {
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Running...");
            }
        }).start();
    }
}
```

Sleep and Join

- Sleep: To wait for a specific amount of time.
- Join: Wait until the Thread finishes.

```
public class SimpleThread {

    static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000L);
                    System.out.println("Running 1st Thread");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });


        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000L);
                    System.out.println("Running 2nd Thread");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println("Ended....");
    }
}
```

Thread Pool

- A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface **ExecutorService** and the class **ThreadPoolExecutor**

- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform

- To use thread pools, we first create a object of **ExecutorService** and pass a set of tasks/runnable to it. **ThreadPoolExecutor** class allows to set the core and maximum pool size.

Thread Pool (cont.)

SingleThreadExecutor

- With a single thread executor results are guaranteed to be executed in the order in which they are added to the executor service.

- It is important to call shutdown otherwise your application will not terminate.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class SingleExecutorServiceDemo {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        try {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    System.out.println("Thread 1");
                }
            });
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    System.out.println("Thread 2");
                }
            });
        } finally {
            executorService.shutdown();
        }
        System.out.println("End");
    }
}
```

Thread Pool (cont.)

FixedThreadPool

- Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable{
  int id;
  public Task(int id) {
    this.id = id;
  }
  @Override
  public void run() {
    System.out.println("Thread name::"+Thread.currentThread().getName()+" Start :"+id);
    try {
      Thread.sleep(2000L);
    } catch (InterruptedException ignored) {
    }

    System.out.println("Thread name::"+Thread.currentThread().getName()+" End :"+id);
  }
}

class ExecutorServiceDemo {
  public static void main(String[] args) {
    ExecutorService executorService= Executors.newFixedThreadPool(2);
    for (int i = 0; i <= 10; i++) {
      executorService.submit(new Task(i));
    }
    executorService.shutdown();
  }
}
```

Thread Pool (cont.)

CachedThreadPool

- Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Process implements Runnable{
    int id;
    public Process(int id) {
        this.id = id;
    }
    @Override
    public void run() {
        System.out.println("Thread name::"+Thread.currentThread().getName()+" Start :"+id);
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException ignored) {
        }

        System.out.println("Thread name::"+Thread.currentThread().getName()+" End :"+id);
    }
}

class ExecutorServiceDemo {
    public static void main(String[] args) {
        ExecutorService executorService= Executors.newCachedThreadPool();
        for (int i = 0; i <= 10; i++) {
            executorService.submit(new Process(i));
        }
        executorService.shutdown();
    }
}
```

Thread Pool (cont.)

ScheduledThreadPool

- Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

class ExecutorServiceSchedulingTasks {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ScheduledExecutorService executorService = Executors.newSingleThreadScheduledExecutor();
        executorService
                .scheduleWithFixedDelay(new Runnable() {
                        @Override
                        public void run() {
                            System.out.println("Task executed after 10 second");
                        }
                      },
                     1,
                     10,
                     TimeUnit.SECONDS);
    }
}
```

Thread Pool (cont.)

ScheduledThreadPool

- Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

class ExecutorServiceSchedulingTasks {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ScheduledExecutorService executorService = Executors.newSingleThreadScheduledExecutor();
        executorService
                .scheduleWithFixedDelay(new Runnable() {
                        @Override
                        public void run() {
                            System.out.println("Task executed after 10 second");
                        }
                    },
                    1,
                    10,
                    TimeUnit.SECONDS);
    }
}
```

Wait and NotifyAll

```
public class WaitAndNotifyAll {

    public void worker1() {
        synchronized (this) {
            System.out.println("Worker1 Started");
            try {
                wait();
            } catch (InterruptedException ignored) {
            }
            System.out.println("Worker1 Done");
        }
    }

    public void worker2() {
        synchronized (this) {
            System.out.println("Worker 2 Started");
            try {
                wait();
            } catch (InterruptedException ignored) {
            }
            System.out.println("Worker 2 Done");
        }
    }

    public void worker3() {
        synchronized (this) {
            System.out.println("Worker 4 Started");
            System.out.println("Worker 4 Done");
            notifyAll();
        }
    }

    public static void main(String[] args) {
        WaitAndNotifyAll demo = new WaitAndNotifyAll();
        new Thread(new Runnable() {
            @Override
            public void run() {
                demo.worker1();
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                demo.worker2();
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                demo.worker3();
            }
        }).start();
    }
}
```

Deep Dive

- Queue and its implementations
- Synchronization
- Wait and notify/notifyAll
- Thread life cycle methods

Assignment

1. What is the Java Collections Framework? Explain its advantages.

2. List the differences between Vector, ArrayList, LinkedList, and HashSet.

3. Find the First Non-Repeating Character

4. Given an array of integers and a target sum, return the indices of the two numbers that add up to the target

5. You are given two sorted ArrayLists<Integer>. Merge them into a single sorted list.

6. Write a Java program with two threads: Thread 1 prints table of number 2. Thread 2 prints table of number 4. Threads should be synced to print output one by one.

7. Write above program using ExecutorService

CloudKeeper

Thank You