19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Concurrency in Java

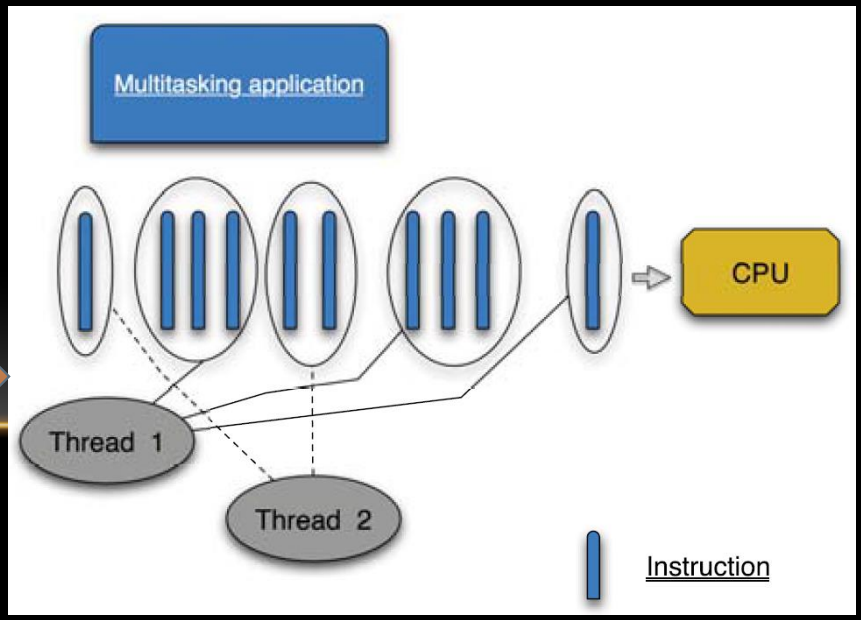
CONCURRENCY

- Ability to run several programs or several parts of a program in parallel.
- A time consuming task can be performed asynchronously or in parallel
- Improves the throughput and the interactivity of the program
- A modern computer has several CPU's or several cores within one CPU.
- The ability to leverage these multi-cores can be the key for a successful high-volume application.

CONCURRENCY

- When more than one task can start and complete in overlapping time periods
- Need not be running in the same instant
- Concurrent programs can be written on a single CPU too
- Multiple tasks are executed in a time-slice manner, where a scheduler (such as the JVM) will guarantee each process a regular “slice” of operating time (Implemented using threads)
- An Illusion of parallelism to users

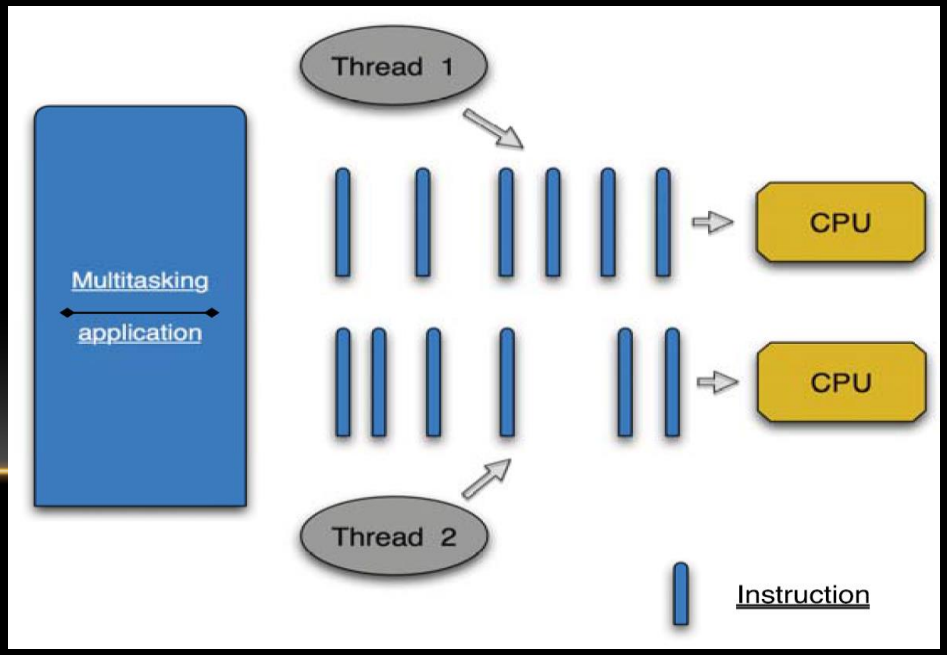
A concurrent application running in a single CPU core with two threads



CONCURRENCY VS PARALLEL PROGRAMMING

- In parallel programming), multiple tasks can be run at the same time, (possible with multicore processors)
- A concurrent program sometimes becomes a parallel program when it's running in a multicore environment.
- *Distributed computing:* multiple computing nodes (computers, virtual machines) spanned across the network, working together on a given problem. A parallel process could be a distributed process when it's running on multiple network nodes.

A concurrent and parallel application running in a two-CPU core with two threads. Both threads are running at the same time.



PROCESS VS THREADS

Process	Thread
Runs independently and isolated of other processes	A thread is a so called lightweight process
Cannot directly access shared data in other processes	Has its own call stack, but can access shared data of other threads in the same process
The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.	Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache. A thread can re-read the shared data

- A Java application runs by default in one process.
- Within a Java application you work with several threads to achieve parallel processing or asynchronous behaviour.

CONCURRENCY ISSUES – SAFETY, LIVENESS, FAIRNESS

- Threads have their own call stack, but can also access shared data.
- Therefore there are two basic problems of visibility and access.
- Visibility problem: occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.
- Access problem: can occur if several threads access and change the same shared data at the same time.
- Visibility and access problem can lead to:
  - Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.
  - Safety failure: The program creates incorrect data.
- If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

CAUSES OF STARVATION IN JAVA

- Threads with high priority swallow all CPU time from threads with lower priority.
- Threads are blocked indefinitely waiting to enter a synchronized block, because other threads are constantly allowed access before it.
- Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it

CHALLENGES WITH CONCURRENT PROGRAMMING

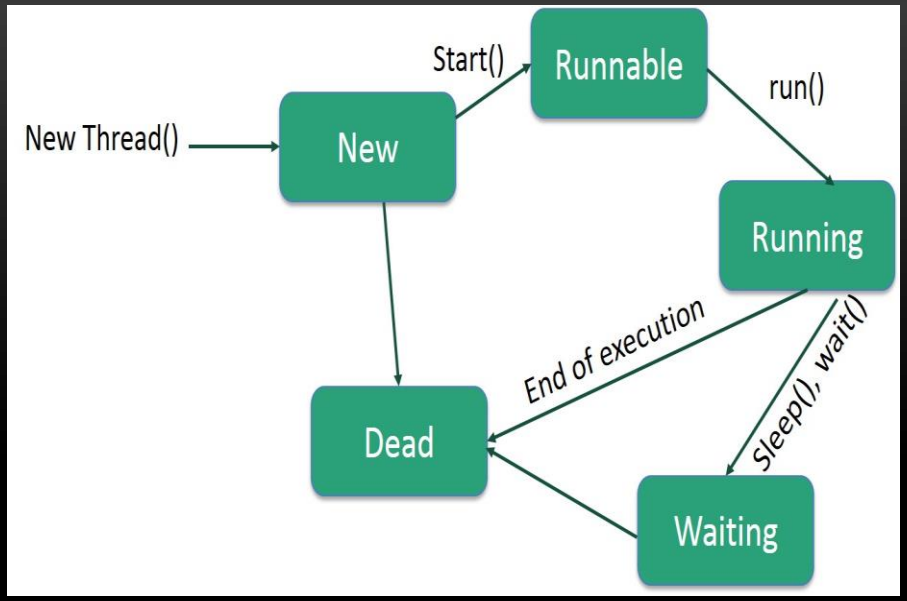
- Writing a correct, concurrent application or program. The *correctness* of the program is important.
- Debugging multithreaded programs is difficult. The same program that causes deadlock in production might not have any locking issues when debugging locally. Sometimes threading issues show up after years of running in production.
- Threading encourages shared state concurrency, and it's hard to make programs run in parallel because of locks, semaphores, and dependencies between threads.

MULTI-TASKING VS MULTI-THREADING

- Multitasking is when multiple processes share common processing resources such as a CPU
- Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
  - Each of the threads can run in parallel.
  - The OS divides processing time not only among different applications, but also among each thread within an application.
  - Enables you to write in a way where multiple activities can proceed concurrently in the same program.
- Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java.

LIFE CYCLE OF A THREAD

- **New – A** thread begins its life cycle in this state
- **Runnable –** After a New thread is started by the program, it becomes runnable
- **Waiting** – A thread waits for another thread to perform a task. The thread transitions back to the runnable state only when the other thread signals the waiting thread to continue executing.
- **Timed Waiting -** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead) –** A runnable thread enters the terminated state when it completes its task or terminates otherwise.



THREAD PRIORITIES

- Java thread priorities range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10).
- By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.
- However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

CREATE A THREAD BY IMPLEMENTING A RUNNABLE INTERFACE (TO EXECUTE YOUR CLASS AS A THREAD)

- **Step 1**
  - Implement public void run( ) method provided by runnable interface
  - This method provides an entry point for the thread and hence the complete execution logic is put inside this method.
- **Step 2**
  - Instantiate a **Thread** object using the following constructor: Thread(Runnable threadObj, String threadName);

  (threadObj is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread)
- **Step 3**

  Once a Thread object is created, you can start it by calling **void start()** method, which executes a call to run( ) method.

EXAMPLE TO CREATE A NEW THREAD AND RUN IT

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo(String name) {
      threadName = name;
      System.out.println("Creating " +
threadName );
   }

   public void run() {
      System.out.println("Running " +
threadName );

      try {

         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " +
threadName + ", " + i);
          // Let the thread sleep for a while.
            Thread.sleep(50);
          }
       } catch (InterruptedException e) {
          System.out.println("Thread " +
threadName + " interrupted.");
       }
       System.out.println("Thread " +  threadName
+ " exiting.");
    }

    public void start () {
       System.out.println("Starting " +
threadName );

       if (t == null) {
          t = new Thread (this, threadName);
          t.start ();
       }
    }
}
```

TESTING THE THREAD AND OUTPUT

```
public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo("Thread-
1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo("Thread-
2");
      R2.start();
   }
}
```

```
D:\PPL\Java>javac TestThread.java

D:\PPL\Java>java TestThread
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Running Thread-2
Thread: Thread-1, 4
Thread: Thread-2, 4
Thread: Thread-2, 3
Thread: Thread-1, 3
Thread: Thread-2, 2
Thread: Thread-1, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

CREATE A THREAD BY EXTENDING A THREAD CLASS

- Create a new class that extends Thread class by following two simple steps
- The approach provides more flexibility in handling multiple threads created using available methods in Thread class.
- **Step 1**
  - Override public void run( ) method available in Thread class which provides an entry point for the thread and put the complete computation logic in this method
- **Step 2**
  - Once Thread object is created, start it by calling **void start()** method, which executes a call to run( ) method.

EXAMPLE TO CREATE A NEW THREAD AND RUN IT

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo(String name) {
      threadName = name;
      System.out.println("Creating " +
threadName );
   }

   public void run() {
      System.out.println("Running " +
threadName );

      try {

         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " +
threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +
threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName
+ " exiting.");
   }

   public void start () {
      System.out.println("Starting " +
threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

TESTING THE THREAD AND OUTPUT

```
public class TestThread1{

   public static void main(String args[]) {
      ThreadDemo T1 = new ThreadDemo("Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo("Thread-2");
      T2.start();
   }
}
```

```
D:\PPL\Java>javac TestThread1.java

D:\PPL\Java>java TestThread1
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Running Thread-2
Thread: Thread-1, 4
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-2 exiting.
Thread Thread-1 exiting.
```

OPERATIONS ON THREADS

Sr.No.	Method	Description
1	public void suspend()	This method puts a thread in the suspended state and can be resumed using resume() method.
2	public void stop()	This method stops a thread completely.
3	public void resume()	This method resumes a thread, which was suspended using suspend() method.
4	public void wait()	Causes the current thread to wait until another thread invokes the notify().
5	public void notify()	Wakes up a single thread that is waiting on this object's monitor.

EXAMPLE

```
class ThreadOps implements Runnable {
  public Thread t;
  private String threadName;
  boolean suspended = false;

  ThreadOps(String name) {
    threadName = name;
    System.out.println("Creating " +  threadName );
  }

  public void run() {
    System.out.println("Running " +  threadName );

    try {

      for(int i = 10; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " +
i);

        // Let the thread sleep for a while.
        Thread.sleep(300);

        synchronized(this) {

          while(suspended) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + "
interrupted.");
    }
    System.out.println("Thread " +  threadName + "
exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );

    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }

  void suspend() {
    suspended = true;
  }

  synchronized void resume() {
    suspended = false;
    notify();
  }
}
```

EXAMPLE

```
public class OpsTst {

  public static void main(String args[]) {
    ThreadOps R1 = new ThreadOps("Thread-1");
    R1.start();

    ThreadOps R2 = new ThreadOps("Thread-2");
    R2.start();

    try {
      Thread.sleep(1000);
      R1.suspend();
      System.out.println("Suspending First Thread");
      Thread.sleep(1000);
      R1.resume();
      System.out.println("Resuming First Thread");

      R2.suspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      R2.resume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    } try {
      System.out.println("Waiting for threads to finish.");
      R1.t.join();
      R2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

EXAMPLE

```
D:\PPL\Java>javac OpsTst.java

D:\PPL\Java>java OpsTst
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 10
Running Thread-2
Thread: Thread-2, 10
Thread: Thread-2, 9
Thread: Thread-1, 9
Thread: Thread-2, 8
Thread: Thread-1, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Thread: Thread-1, 6
Suspending thread Two
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```

THREAD POOLS (TO EXECUTE TASKS EFFICIENTLY)

- Single task execution using **java.lang.Runnable** is not efficient for a large number of tasks
- A thread has to be created for each task
  - Could limit throughput and cause poor performance.
- *thread pool* is an ideal way to manage the number of tasks executing concurrently.
  - **Executor** interface for executing tasks in a thread pool and the
  - **ExecutorService** interface for managing and controlling tasks in Java

SYNCHRONIZATION AND LOCKS

- A shared resource may become corrupted if it is accessed simultaneously by multiple threads.

- Example:
  - Consider an application that creates and launches 100 threads
  - Each thread adds a rupee to an account

- Classes Required for the above application:
  - Define a class named Account to model the account,
  - a class named AddAPennyTask to add a penny to the account, and
  - a main class that creates and launches threads.

```
import java.util.concurrent.*;
public class AccountWithoutSync {
private static Account account = new Account();

public static void main(String[] args) {
ExecutorService executor = Executors.newCachedThreadPool(); //create
executor

// Create and launch 100 threads
for (int i = 0; i < 100; i++) {
    executor.execute(new AddAPennyTask());  //submit task
  }

  executor.shutdown();                //shut down executor

  // Wait until all tasks are finished
  while (!executor.isTerminated()) {     //wait for all tasks to
//terminate
  }

  System.out.println("What is balance? " + account.getBalance());
  }

 // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
  public void run() {
  account.deposit(1);
  }
  }

  // An inner class for account
  private static class Account {
  private int balance = 0;

  public int getBalance() {
  return balance;
  }

  public void deposit(int amount) {
  int newBalance = balance + amount;

  // This delay is deliberately added to magnify the
  // data-corruption problem and make it easy to see.
  try {
  Thread.sleep(5);
  }
  catch (InterruptedException ex) {
  }

  balance = newBalance;
  }
  }
  }
```

```
D:\PPL\Java>java AccountWithoutSync
What is balance? 3

D:\PPL\Java>java AccountWithoutSync
What is balance? 2

D:\PPL\Java>java AccountWithoutSync
What is balance? 3

D:\PPL\Java>java AccountWithoutSync
What is balance? 2

D:\PPL\Java>java AccountWithoutSync
What is balance? 2

D:\PPL\Java>java AccountWithoutSync
What is balance? 3
```

RACE CONDITION IN MULTITHREADED PROGRAMS

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

- Task 1 and Task 2 both add 1 to the same balance
- In Step 1, Task 1 gets the balance from the account.
- In Step 2, Task 2 gets the same balance from the account.
- In Step 3, Task 1 writes a new balance to the account.
- In Step 4, Task 2 writes a new balance to the account.
- The effect of this scenario is that Task 1 does nothing because in Step 4
- Task 2 overrides Task 1's result.
- The problem is that Task 1 and Task 2 are accessing a common resource in a way that causes a conflict.

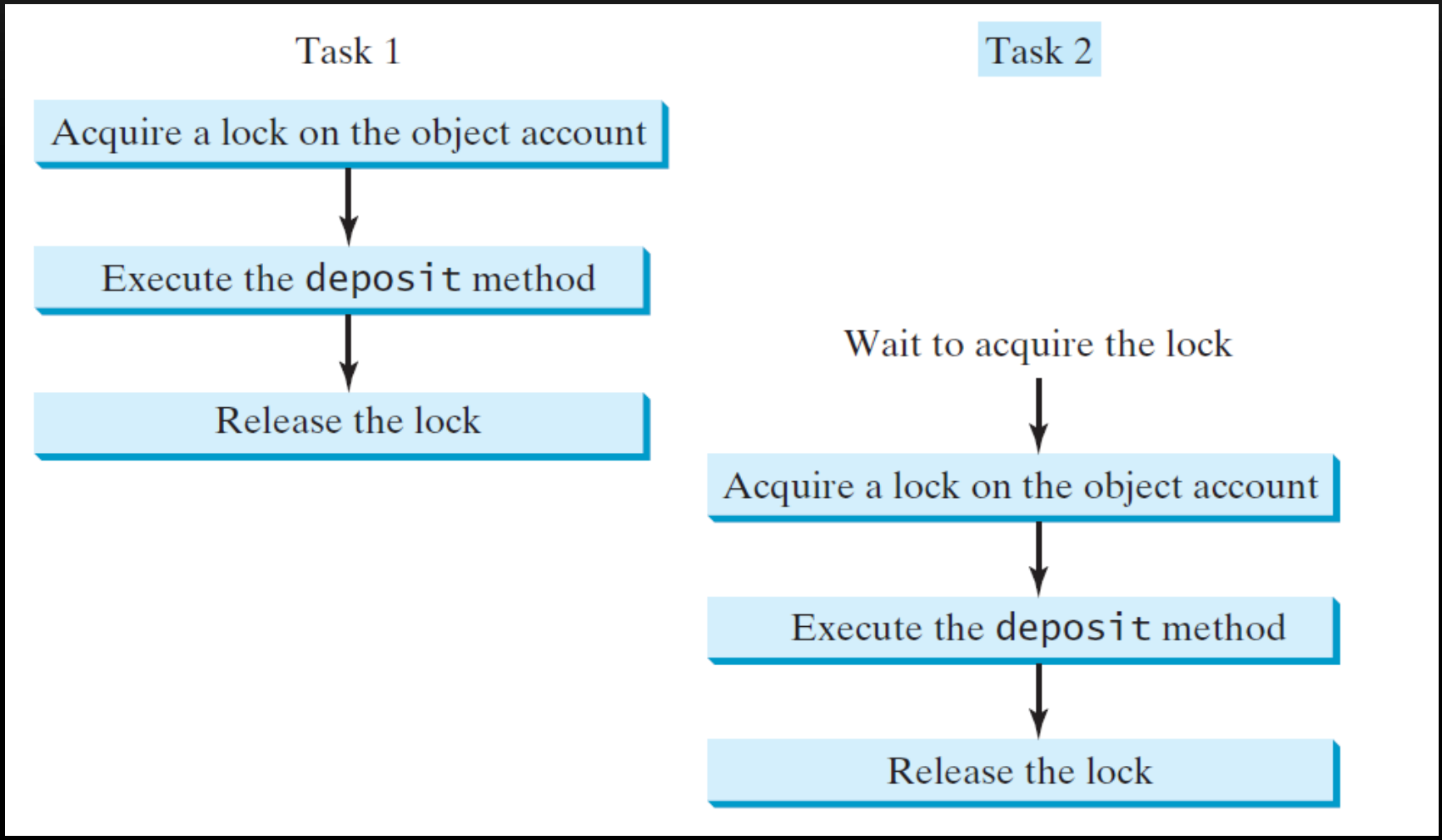
THE **SYNCHRONIZED** KEYWORD

- Used to access the critical region (E.g., deposit method) by only one thread at a time using

  public synchronized void deposit(**double** amount)

- A synchronized method acquires a lock before it executes.
- Lock
  - A mechanism for exclusive use of a resource
  - For instance method, the lock is on the object for which the method was invoked
  - For static method, the lock is on the class
  - If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released.
  - Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

DEPOSIT AFTER SYNCHRONISATION



SYNCHRONISED STATEMENT / BLOCK

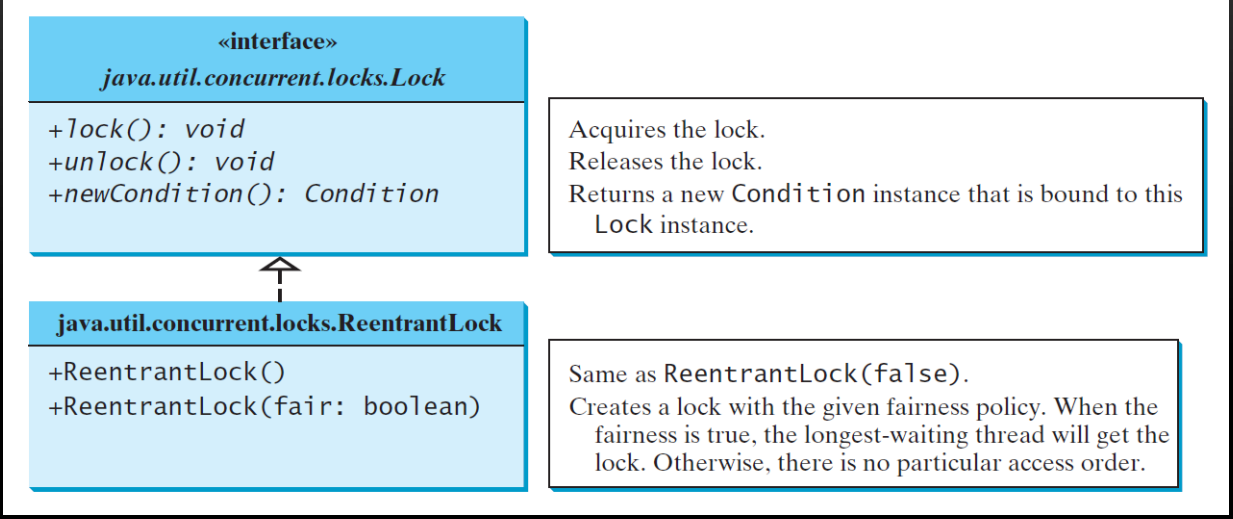
```
synchronized (expr) {
statements;
}
```

- The expression **expr** must evaluate to an object reference.
- Example:

```
synchronized (account) {
account.deposit(1);
}
```

- If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed and then the lock is released.

SYNCHRONIZATION USING LOCKS IN JAVA



- Lock: instance of the **Lock** interface - defines the methods for acquiring and releasing locks
- **newCondition()** method: to create any number of **Condition** objects - can be used for thread communications
- **ReentrantLock:** concrete implementation of **Lock -** creating mutually exclusive locks.
- Fairness Policy:
  - True – guarantees that the longest-waiting thread will obtain the lock first.
  - False – grants a lock to a waiting thread arbitrarily.
- Programs using fair locks accessed by many threads may have poorer overall performance than those using the default setting, but they have smaller variances in times to obtain locks and prevent starvation.

```
import java.util.concurrent.*;
 import java.util.concurrent.locks.*;

 public class AccountWithSyncUsingLock {
 private static Account account = new Account();

 public static void main(String[] args) {
 ExecutorService executor = Executors.newCachedThreadPool();

  // Create and launch 100 threads
  for (int i = 0; i < 100; i++) {
  executor.execute(new AddAPennyTask());
  }
  executor.shutdown();
  // Wait until all tasks are finished
  while (!executor.isTerminated()) {
  }
    System.out.println("What is balance? " + account.getBalance());
  }
  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
  public void run() {
  account.deposit(1);
  }
 }
  // An inner class for Account
  public static class Account {
  private static Lock lock = new ReentrantLock(); // Create a lock
  private int balance = 0;

  public int getBalance() {
  return balance;
  }

  public void deposit(int amount) {
  lock.lock(); // Acquire the lock
    try {
    int newBalance = balance + amount;

    // This delay is deliberately added to magnify the
    // data-corruption problem and make it easy to see.
    Thread.sleep(5);

    balance = newBalance;
    }
    catch (InterruptedException ex) {
    }
    finally {
    lock.unlock(); // Release the lock
    }
   }
  }
 }
```

```
D:\PPL\Java>javac AccountWithSyncUsingLock.java

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100

D:\PPL\Java>java AccountWithSyncUsingLock
What is balance? 100
```

FUTURE AND CALLABLE

- Runnable interface can only run the thread
- In contrast, java.util.concurrent.Callable object can return the computed result done by a thread
- Callable object:
  - returns Future object which provides methods to monitor the progress of a task being executed by a thread.
- Future object:
  - can be used to check the status of a Callable and then retrieve the result from the Callable once the thread has completed execution
  - Also provides timeout functionality

FUTURE AND CALLABLE - SYNTAX

```
//submit the callable using ThreadExecutor and get the result as a Future object

Future<Long> result10 = executor.submit(new FactorialService(10));

//get the result using get method of the Future object

//get method waits till the thread execution and then return the result of the execution.

Long factorial10 = result10.get();
```

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class FutCall {
  public static void main(final String[] arguments)
throws InterruptedException,
    ExecutionException {
    ExecutorService executor =
Executors.newSingleThreadExecutor();
    System.out.println("Factorial Service called
for 10!");
    Future<Long> result10 = executor.submit(new
FactorialService(10));
    System.out.println("Factorial Service called
for 20!");
    Future<Long> result20 = executor.submit(new
FactorialService(20));
    Long factorial10 = result10.get();
    System.out.println("10! = " + factorial10);
    Long factorial20 = result20.get();
    System.out.println("20! = " + factorial20);
    executor.shutdown();
  }
```

```
  static class FactorialService implements
Callable<Long> {
    private int number;
    public FactorialService(int number) {
      this.number = number;
    }
    @Override
    public Long call() throws Exception {
      return factorial();
    }

    private Long factorial() throws
InterruptedException {
      long result = 1;
      while (number != 0) {
        result = number * result;
        number--;
        Thread.sleep(100);
      }
      return result;
    }
  }
```

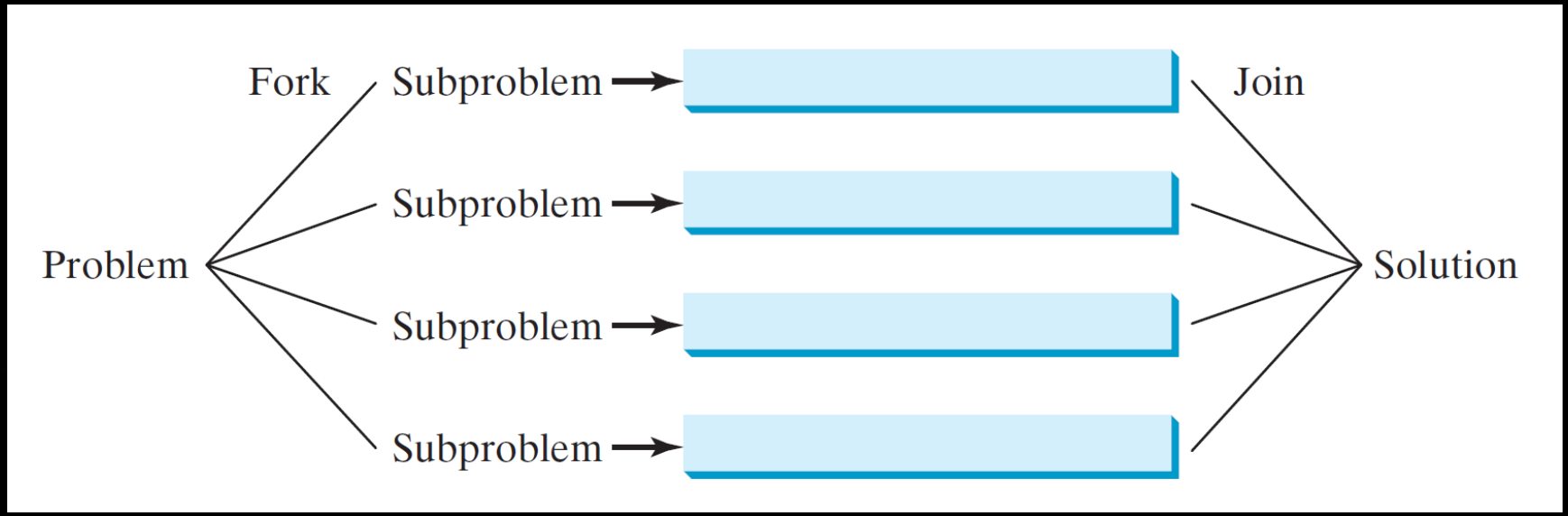
EXAMPLE

EXAMPLE - OUTPUT

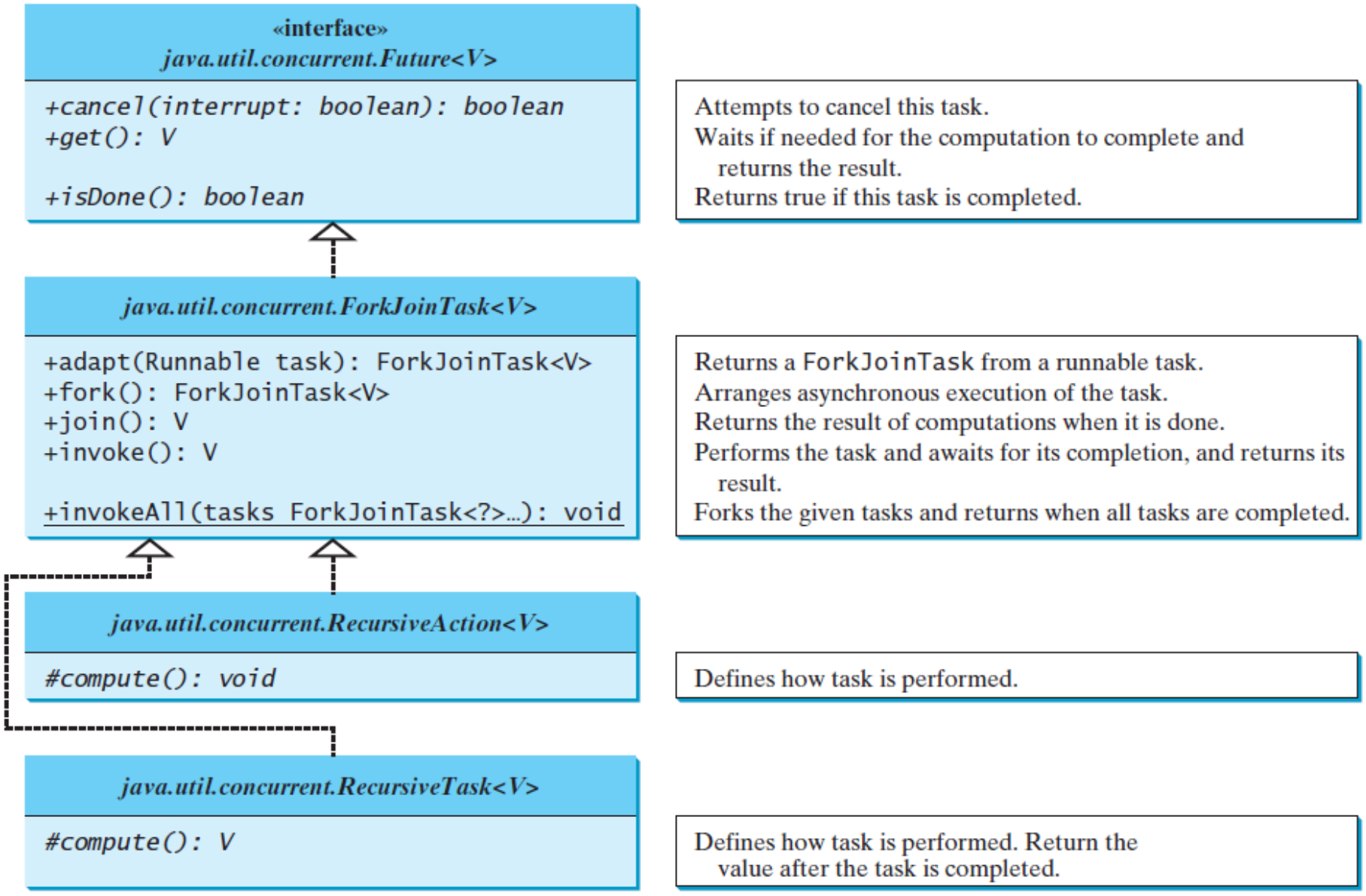
```
D:\PPL\Java>javac FutCall.java

D:\PPL\Java>java FutCall

Factorial Service called for 10!

Factorial Service called for 20!

10! = 3628800

20! = 2432902008176640000
```

PARALLEL PROGRAMMING

- *The Fork/Join Framework is used for parallel programming in Java.*
- A problem is divided into non-overlapping sub-problems, which can be solved independently in parallel.
- A *fork* can be viewed as an independent task that runs on a thread
- The solutions to all sub-problems are then joined to obtain an overall solution for the problem.
- A parallel implementation of the divide-and-conquer approach.



FORK-JOIN TASK (DEFINES A TASK FOR ASYNCHRONOUS EXECUTION)



FORK-JOINTASK CLASS

- Abstract base class for tasks
- Is a thread-like entity, but it is much lighter than a normal thread because huge numbers of tasks and subtasks can be executed by a small number of actual threads in a **ForkJoinPool**
- Tasks are primarily coordinated using **fork()** and **join()**.
- Invoking **fork()** on a task arranges asynchronous execution
- Invoking **join()** waits until the task is completed
- **invoke()** and **invokeAll(tasks)** methods implicitly invoke **fork()** to execute the task and **join()** to wait for the tasks to complete, and return the result, if any
- The static method **invokeAll** takes a variable number of **ForkJoinTask** arguments using the **...** syntax

FORK-JOIN FRAMEWORK

- Fork-join framework
  - designed to parallelize naturally recursive divide-and-conquer solutions
  - allows to break a certain task on several workers and then wait for the result to combine them.
  - It leverages multi-processor machine's capacity to great extent.
- Fork
  - a process in which a task splits itself into smaller and independent sub-tasks which can be executed concurrently
  - Syntax:

```
Sum left = new Sum(array, low, mid);
left.fork();
```

where Sum is a subclass of RecursiveTask and left.fork() splits the task into sub-tasks.

FORK-JOIN FRAMEWORK

- **Join**
  - A process in which a task join all the results of sub-tasks once the subtasks have finished executing, otherwise it keeps waiting.
  - Syntax:

```
left.join();
```

were left is an object of Sum class.

FORKJOINPOOL

- a special thread pool designed to work with fork-and-join task splitting
  - Syntax

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

Here a new ForkJoinPool is created with a parallelism level of 4 CPUs.

RECURSIVE ACTION AND RECURSIVE TASK

- Two subclasses of ForkJoinTask
- To define a concrete task class, your class should extend **RecursiveAction** or **RecursiveTask**
- Your task class should override the **compute()** method to specify how a task is performed.
- Recursive Action:
  - represents a task which does not return any value
  - Syntax

```
class Writer extends RecursiveAction {
@Override
protected void compute() { }
}
```

- Recursive Task:
  - represents a task which returns a value
  - Syntax

```
class Sum extends RecursiveTask<Long> {
@Override protected Long compute() { return null; }
}
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class ForkJoin {
  public static void main(final String[] arguments)
throws InterruptedException,
    ExecutionException {
    int nThreads =
Runtime.getRuntime().availableProcessors();
    System.out.println(nThreads);
    int[] numbers = new int[1000];
    for(int i = 0; i < numbers.length; i++) {
      numbers[i] = i;
    }
    ForkJoinPool forkJoinPool = new
ForkJoinPool(nThreads);
    Long result = forkJoinPool.invoke(new
Sum(numbers,0,numbers.length));
    System.out.println(result);
  }
  static class Sum extends RecursiveTask<Long>
{
    int low;
    int high;
    int[] array;
    Sum(int[] array, int low, int high) {
      this.array = array;
      this.low   = low;
      this.high  = high;
    }
    protected Long compute() {
      if(high - low <= 10) {
        long sum = 0;
        for(int i = low; i < high; ++i)
          sum += array[i];
          return sum;
      } else {
        int mid = low + (high - low) / 2;
        Sum left  = new Sum(array, low, mid);
        Sum right = new Sum(array, mid, high);
        left.fork();
        long rightResult = right.compute();
        long leftResult  = left.join();
        return leftResult + rightResult;
      }
    }
  }
}
```

EXAMPLE

EXAMPLE - OUTPUT

```
D:\PPL\Java>java ForkJoin
4
499500
```

THANK YOU