

# Professional Programming in Java

Session: 13

## Advanced Concurrency and Parallelism





- ◆ Explain the enhancements of `java.util.concurrent` package
- ◆ Describe atomic operations with the new set of classes of the `java.util.concurrent.atomic` package
- ◆ Explain the `StampedLock` class to implement locks
- ◆ Explain the new features of `ForkJoinPool`
- ◆ Define parallel streams
- ◆ Describe parallel sorting of arrays
- ◆ Identify recursive actions of the fork/join framework

For Aptech Center Use Only

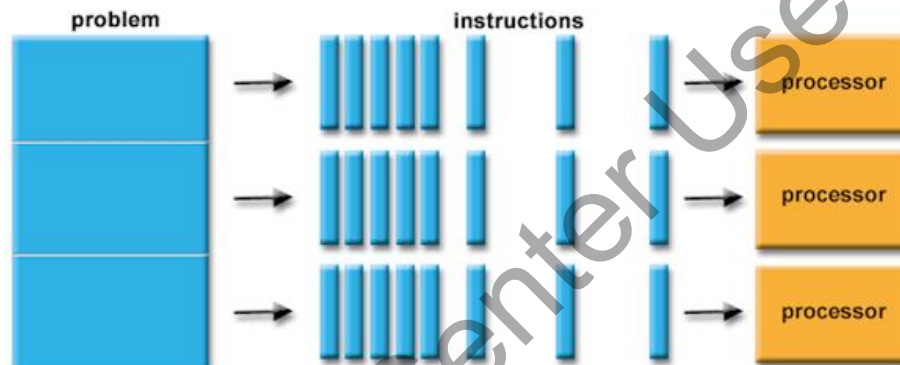


- ◆ Similar to tasks that run in parallel in real world, parallelization refers to the process of taking a serial code that runs on a single CPU and spreading the work across multiple CPUs.

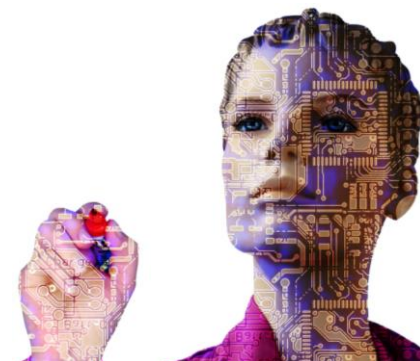




- ◆ An important enhancement in Java programming language is parallelization.



- ◆ Parallelization is used to make applications run efficiently.
- ◆ Challenges faced by parallelism are addressed through work stealing strategy.
- ◆ The Fork-Join framework in Java meets the work stealing requirements through recursive job partitioning.



# New Enhancements in the `java.util.concurrent` Package



- ◆ Many new enhancements have been made to the `java.util.concurrent` package

## Classes

- `CompletableFuture`
- `CountedCompleter`
- `ConcurrentHashMap.KeySetView`

## Interface

- `CompletableFuture.AsynchronousCompletionTask`
- `CompletionStage<T>`

## Exception

- `CompletionException`



- ◆ The class implements the `CompletionStage` and the `Future` interface to simplify asynchronous operations.
- ◆ An existing `Future` interface represents the result of an asynchronous computation.
- ◆ The `get()` method of the `Future` interface returns the result of the computation.
- ◆ The methods of the `CompletableFuture` class run asynchronously without stopping the program execution.







## Methods of the CompletableFuture Class

Method	Description
<code>supplyAsync()</code>	Accepts a <code>Supplier</code> object that contains code to be executed asynchronously
<code>thenApply()</code>	Returns a new <code>CompletableFuture</code> object that is executed with the result of the completed stage, provided the current stage completes normally
<code>join()</code>	Returns the result when the current asynchronous computation completes, or throws an exception of type <code>CompletionException</code>
<code>thenAccept()</code>	Accepts a <code>Consumer</code> object
<code>whenComplete()</code>	Uses <code>BiConsumer</code> as an argument
<code>getNow()</code>	Sets the value passed to it as the result if the calling completion stage is not completed

# CountedCompletor Class [1-8]



- ◆ The class extends `ForkJoinTask` to complete an action performed when triggered, provided there are no pending actions.

The `tryComplete()` method checks if the pending count is nonzero, and if so, decrements the count.

Or, the `tryComplete()` method invokes the `onCompletion(CountedCompleter)` method and attempts to complete the task, and if successful, marks this task as complete.

The `compute()` method performs the main computation and invokes the `tryComplete()` method.





- ◆ The `CountedCompletor` class may override the following methods:

Method	Description
<code>onCompletion(CountedCompleter)</code>	To perform some action upon normal completion.
<code>onExceptionalCompletion(Throwable, CountedCompleter)</code>	To perform some action when an exception is thrown.

For Aptech Center Use Only



- ◆ The `CountedCompleter<Void>` class is declared as `CountedCompleter<Void>` when the class does not generate results.
- ◆ The class returns `null` as a value.
- ◆ The class must be overridden to yield a value.





- ◆ The code demonstrates the implementation of the CountedCompleter class.

## Code Snippet

```
package com.training.demo.countedcompleter;
import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.CountedCompleter;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;

public class CountedCompleterDemo {
    static class NumberComputator extends
        CountedCompleter<Void> {
        final ConcurrentLinkedQueue<String>
            concurrentLinkedQueue;
        final int start;
        final int end;
```



```
NumberComputator(ConcurrentLinkedQueue<String>
concurrentLinkedQueue, int start, int end) {
this(concurrentLinkedQueue, start, end, null);
}

NumberComputator(ConcurrentLinkedQueue<String>
concurrentLinkedQueue, int start, int end,
NumberComputator parent) {
super(parent);
this.concurrentLinkedQueue = concurrentLinkedQueue;
this.start = start;
this.end = end;
}

@Override
public void compute() {
if (end - start < 5) {
String s = Thread.currentThread().getName();
for (int i = start; i < end; i++) {
concurrentLinkedQueue.add(String.format("Iteration
number: {%d} performed by thread {%s}", i, s));
}
```



```
}
propagateCompletion();
} else {
    int mid = (end + start) / 2;
    NumberComputator subTaskA = new
    NumberComputator(concurrentLinkedQueue, start,
    mid, this);
    NumberComputator subTaskB = new
    NumberComputator(concurrentLinkedQueue, mid, end,
    this);
    setPendingCount(1);
    subTaskA.fork();
    subTaskB.compute();
}
}
}

public static void main(String[] args) throws
    ExecutionException, InterruptedException {
    ConcurrentLinkedQueue<String> linkedQueue = new
```

# CountedCompletor Class [7-8]



```
ConcurrentLinkedQueue<>();  
NumberComputator numberComputator = new  
ForkJoinPool.commonPool().invoke(numberComputator);  
ArrayList<String> list = new  
ArrayList<>(linkedQueue);  
for (String listItem : list) {  
    System.out.println(" " + listItem);  
}  
}  
}
```

For Aptech Center Use Only

# CountedCompletor Class [8-8]



Following is the output of the code:

```
Output - CountedCompletorDemo (run) X
run:
Iteration number: {44} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {53} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {97} performed by thread {main}
Iteration number: {54} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {45} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {98} performed by thread {main}
Iteration number: {46} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {99} performed by thread {main}
Iteration number: {55} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {51} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {42} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {95} performed by thread {main}
Iteration number: {43} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {52} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {96} performed by thread {main}
Iteration number: {62} performed by thread {ForkJoinPool.commonPool-worker-1}
Iteration number: {38} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {63} performed by thread {ForkJoinPool.commonPool-worker-1}
Iteration number: {47} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {39} performed by thread {ForkJoinPool.commonPool-worker-2}
```





- ◆ Provides a view of the keys contained in the class
- ◆ Implements the Set interface and thus, can access the keys as a Set object
- ◆ The Set object and the ConcurrentHashMap object have a bi-directional relationship

For Aptech Center Use Only



- ◆ The code demonstrates the method to access the keys of a ConcurrentHashMap as a Set.

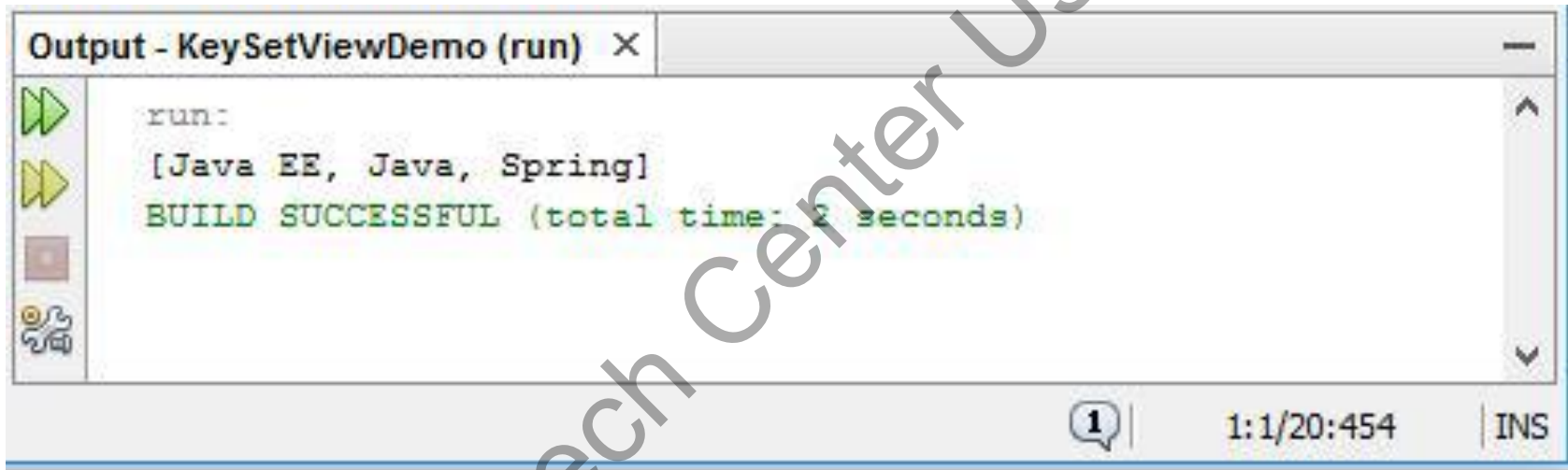
## Code Snippet

```
package com.training.demo.kesyssetview;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import
java.util.concurrent.ConcurrentHashMap;

public class KeySetViewDemo {
    public static void main(String[] args) {
        Map<String,String> map =
            new ConcurrentHashMap<>();
        map.put("Spring", "Spring");
        Set keySet = map.keySet();
        System.out.println(keySet);
    }
}
```



Following is the output of the code:



```
run:
[Java EE, Java, Spring]
BUILD SUCCESSFUL (total time: 2 seconds)
```



New classes in the `java.util.concurrent.atomic` package

`LongAccumulator`: Maintains a long running value updated using a supplied function

`LongAdder`: Maintains an initially zero long sum

`DoubleAccumulator`: Maintains a double running value updated using a supplied function

`DoubleAdder`: Maintains an initially zero double sum



- ◆ The code demonstrates the use of the `LongAdder` and `DoubleAdder` atomic operation classes.

## Code Snippet

```
package com.training.demo.atomicoperation;
import java.util.concurrent.atomic.DoubleAdder;
import java.util.concurrent.atomic.LongAdder;
public class AtomicOperationClassDemo {
    private final LongAdder longAdder;
    private final DoubleAdder doubleAdder;

    public AtomicOperationClassDemo(LongAdder
longAdder, DoubleAdder doubleAdder)
    {
        this.longAdder = longAdder;
        this.doubleAdder = doubleAdder;
    }
}
```



```
public void incrementLong() {
    longAdder.increment();
}
public long getLongCounter() {
    return longAdder.longValue();
}
public void addDouble(int doubleValue) {
    doubleAdder.add(doubleValue);
}
public double getSumAsDouble() {
    return doubleAdder.doubleValue();
}
public static void main(String[] args) {
    AtomicOperationClassDemo
    atomicOperationClassDemo1 = new
    AtomicOperationClassDemo(new LongAdder(), new
    DoubleAdder());
}
```



```
System.out.println("-----Long Counter-----");
for (int i = 0; i < 10; i++) {
    atomicOperationClassDemo1.incrementLong();
    System.out.println("Long Counter " +
        atomicOperationClassDemo1.getLongCounter());
}
System.out.println("-----Double Sum-----");
for (int j = 0; j < 10; j++) {
    atomicOperationClassDemo1.addDouble(j);
    System.out.println("Double Sum " +
        atomicOperationClassDemo1.getSumAsDouble());
}
}
```

For Aptech Center Use Only



# Atomic Operations and Locks [5-15]



Following is the output of the code:

A screenshot of a Java IDE's output window titled "Output - AtomicOperationClassDemo (run) x". The window contains the following text:

```
run:
-----Long Counter-----
Long Counter 1
Long Counter 2
Long Counter 3
Long Counter 4
Long Counter 5
Long Counter 6
Long Counter 7
Long Counter 8
Long Counter 9
Long Counter 10
-----Double Sum-----
Double Sum 0.0
Double Sum 1.0
Double Sum 3.0
Double Sum 6.0
Double Sum 10.0
Double Sum 15.0
Double Sum 21.0
Double Sum 28.0
Double Sum 36.0
Double Sum 45.0
BUILD SUCCESSFUL (total time: 1 second)
```

The output shows a sequence of 10 long counter increments followed by a sequence of 10 double sum calculations. The final output is "BUILD SUCCESSFUL (total time: 1 second)".



- ◆ The `StampedLock` class of the `java.util.concurrent.locks` package enables to implement locks.
- ◆ The class returns a `long` number, also known as stamp, when a lock is granted.
- ◆ The stamp is used either to release a lock or to check if the lock is valid.
- ◆ The lock supports a new lock mode known as optimistic locking.

For Apteck Center Use Only



Implements lock with three modes for read/write access

## Writing

- This mode is achieved through the `writeLock()` method.
- The `writeLock()` method returns a stamp to release a lock.
- This mode is supported by timed and untimed versions of `tryWriteLock()` method.
- When a thread is locked in write mode then no read locks can be obtained.



Implements lock with three modes for read/write access

## Reading

- This mode is achieved through the `readLock()` method.
- The `readLock()` method returns a stamp to release a lock.
- This mode is supported by timed and untimed versions of `tryReadLock()` method.



Implements lock with three modes for read/write access

## Optimistic Reading

- This mode is achieved through the `tryOptimisticRead()` and `validate(long)` methods.
- The `tryOptimisticRead()` returns a non-zero stamp only if the lock is not held in the write mode.
- The `validate(long)` returns the value `true` if the lock has not been acquired by any other thread in the write mode.



- ◆ Code demonstrates use of the `StampedLock` class in reading, writing, and optimistic reading mode.

## Code Snippet

```
package com.training.demo.stampedlock;
import java.util.concurrent.locks.StampedLock;

public class StampedLockDemo {
    private final StampedLock stampedLock = new
        StampedLock();
    private double balance;
    public StampedLockDemo(double balance) {
        this.balance = balance;
        System.out.println("Available balance: "+balance);
    }
    public void deposit(double amount) {
        System.out.println("\nAbout to deposit $:
        "+amount);
        long stamp = stampedLock.writeLock();
        System.out.println("Applied read lock");
    }
}
```



```
try {
    balance += amount;
    System.out.println("Available balance: "+balance);
} finally {
    stampedLock.unlockWrite(stamp);
    System.out.println("Unlocked write lock");
}
}

public void withdraw(double amount) {
    System.out.println("\nAbout to withdraw $: "+amount);
    long stamp = stampedLock.writeLock();
    System.out.println("Applied write lock");
    try {
        balance -= amount;
        System.out.println("Available balance: "+balance);
    } finally {
        stampedLock.unlockWrite(stamp);
        System.out.println("Unlocked write lock");
    }
}
```





```
public double checkBalance() {
    System.out.println("\nAbout to check balance");
    long stamp = stampedLock.readLock();
    System.out.println("Applied read lock");
    try {
        System.out.println("Available balance: "+balance);
        return balance;
    } finally {
        stampedLock.unlockRead(stamp);
        System.out.println("Unlocked read lock");
    }
}

public double checkBalanceOptimisticRead() {
    System.out.println("\nAbout to check balance with
    optimistic read lock");
    long stamp = stampedLock.tryOptimisticRead();
    System.out.println("Applied non-blocking optimistic
    read lock");
    double balance = this.balance;
```



```
if (!stampedLock.validate(stamp)) {
    System.out.println("Stamp have changed. Applying
        full-blown read lock.");
    stamp = stampedLock.readLock();
    try {
        balance = this.balance;
    } finally {
        stampedLock.unlockRead(stamp);
        System.out.println("Unlocked read lock");
    }
}
System.out.println("Available balance: "+balance);
return balance;
}
```

For Aptech Center Use Only

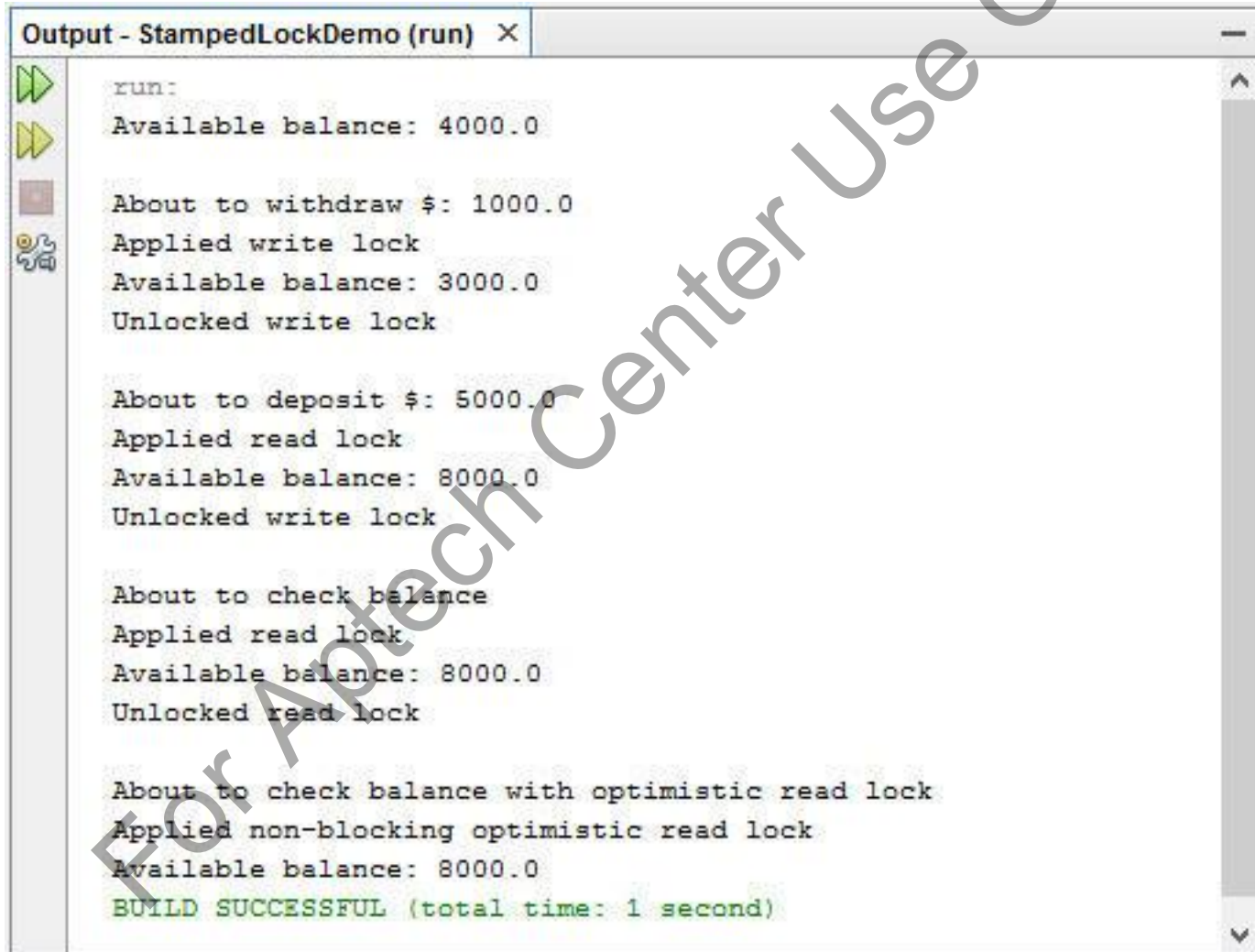


```
public static void main(String[] args){  
    StampedLockDemo stampedLockDemo=new  
    StampedLockDemo(4000.00);  
    stampedLockDemo.withdraw(1000.00);  
    stampedLockDemo.deposit(5000.00);  
    stampedLockDemo.checkBalance();  
    stampedLockDemo.checkBalanceOptimisticRead();  
}
```

For Aptech Center Use Only



Following is the output of the code:

A screenshot of a Java IDE's output window titled "Output - StampedLockDemo (run)". The window shows the execution output of a program. On the left side of the window, there are several icons: a green play button, a yellow play button, a red square, and a bug icon. The output text is as follows:

```
run:
Available balance: 4000.0

About to withdraw $: 1000.0
Applied write lock
Available balance: 3000.0
Unlocked write lock

About to deposit $: 5000.0
Applied read lock
Available balance: 8000.0
Unlocked write lock

About to check balance
Applied read lock
Available balance: 8000.0
Unlocked read lock

About to check balance with optimistic read lock
Applied non-blocking optimistic read lock
Available balance: 8000.0
BUILD SUCCESSFUL (total time: 1 second)
```



- ◆ The framework has included many new features such as:

New  
ForkJoinPool  
Features

Stream  
Parallelization

Array Sorting  
Parallelism

Recursive  
Action



- ◆ Enables the use of a common thread
- ◆ Helps the application to reduce resource usage
- ◆ Methods included in the `ForkJoinPool` class are:

A diagram consisting of two white circles with red outlines, connected by a blue line. The top circle is connected to the top-left of the text box, and the bottom circle is connected to the top-left of the text box below it.

`commonPool()`: A static method that returns a common thread pool.

`getCommonPoolParallelism()`: A new method that returns the targeted parallelism level of the common thread pool.



- ◆ The code demonstrates the use of parallel stream to iterate through the elements of an `ArrayList`.

## Code Snippet

```
package com.training.demo.parallelstream;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class ParallelStreamDemo {
    public static void main(String[] args) {
        List<String> items = new ArrayList<String>();
        items.add("one");
        items.add("two");
        items.add("three");
        items.add("four");
        Stream parallelStream = items.parallelStream();
        parallelStream.forEach(System.out::println);
    }
}
```



# Stream Parallelization [2-2]



Following is the output of the code:

```
Output - ParallelStreamDemo (run) x
run:
three
four
two
one
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only



- ◆ The code demonstrates how to sort arrays in parallel.

## Code Snippet

```
package com.training.demo.parallelarraysort;
import java.util.Arrays;

public class ParallelArraySortDemo {
    public static void main(String[] args) {
        int[] intArray = new int[100];
        for(int i = 0; i < intArray.length; i++) {
            intArray[i] = (int) (Math.random() * 100);
        }
        Arrays.parallelSort(intArray);
        System.out.println(Arrays.toString(intArray));
    }
}
```

# Arrays Sort Parallelism [2-2]



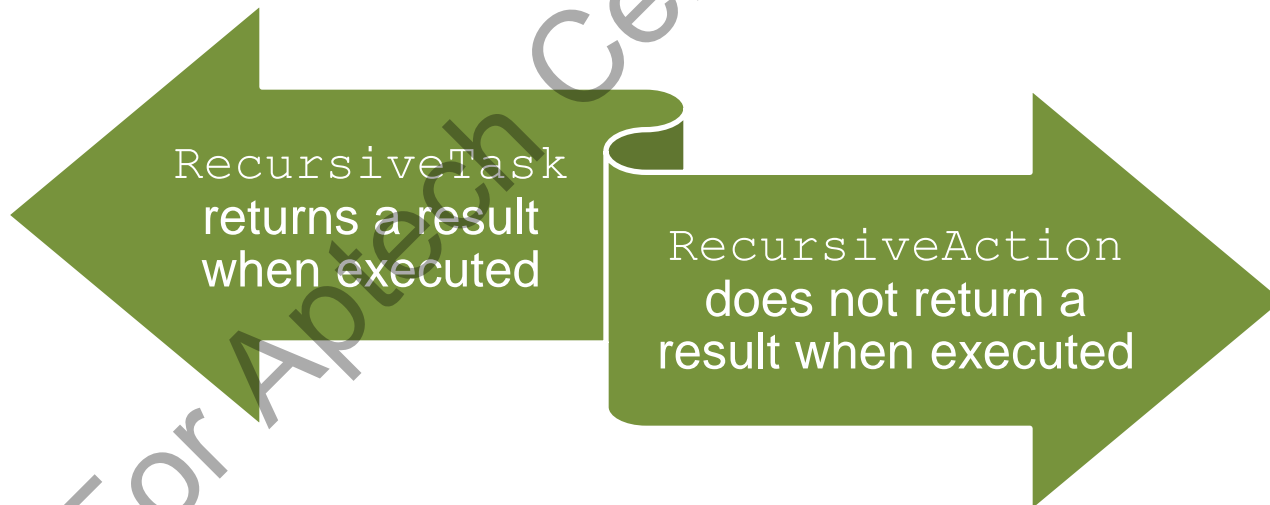
Following is the output of the code:

```
Output - ParallelArraySortDemo (run) X
run:
[0, 2, 5, 5, 6, 7, 8, 9, 10, 11, 11, 16, 16, 17, 19, 20, 21, 22, 22, 22, 24, 24, 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

For Aptech Center Use Only



- ◆ The Fork-Join framework extends `AbstractExecutorService` class to override the `ForkJoinTask` processes.
- ◆ Two new classes have been to implement the `ForkJoinTask` processes:





- ◆ The code demonstrates the use of Fork-Join functionality with RecursiveAction.

### Code Snippet

```
package com.training.demo.recursiveaction;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class RecursiveActionDemo extends RecursiveAction {
    private long assignedWork = 0;
    public RecursiveActionDemo(long assignedWork) {
        this.assignedWork = assignedWork;
    }
    private List<RecursiveActionDemo> createSubtasks() {
        List<RecursiveActionDemo> subtaskList = new
        ArrayList<>();
    }
}
```



```
RecursiveActionDemo subtask1 = new
RecursiveActionDemo(this.assignedWork / 2);
RecursiveActionDemo subtask2 = new
RecursiveActionDemo(this.assignedWork / 2);
subtaskList.add(subtask1);
subtaskList.add(subtask2);
return subtaskList;
}
@Override
protected void compute() {
    if (this.assignedWork > 50) {
        System.out.println("Splitting assignedWork : " +
            Thread.currentThread() + " computing: : " +
            this.assignedWork);
        List<RecursiveActionDemo> subtaskList = new
        ArrayList<>();
        subtaskList.addAll(createSubtasks());
        for (RecursiveAction subtask : subtaskList) {
            subtask.fork();
        }
    }
}
```



```
    }  
  } else {  
    System.out.println("Main thread " +  
      Thread.currentThread() + " computing: : " +  
      this.assignedWork);  
  }  
}  
public static void main(String[] args) {  
    RecursiveActionDemo recursiveActionDemo = new  
    RecursiveActionDemo(500);  
    final ForkJoinPool forkJoinPool = new  
    ForkJoinPool(4);  
    forkJoinPool.invoke(recursiveActionDemo);  
}  
}
```



Following is the output of the code:

```
Output - RecursiveAction (run) ×
run:
Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: : 500
Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: : 250
Splitting assignedWork : Thread[ForkJoinPool-1-worker-2,5,main] computing: : 250
Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: : 125
Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: : 62
Main thread Thread[ForkJoinPool-1-worker-1,5,main] computing: : 31
Main thread Thread[ForkJoinPool-1-worker-1,5,main] computing: : 31
Splitting assignedWork : Thread[ForkJoinPool-1-worker-3,5,main] computing: : 125
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only





- ◆ The `CompletableFuture` class simplifies coordination of asynchronous operations.
- ◆ The `CountedCompletor` class represents a completion action performed when triggered, provided there are no remaining pending actions.
- ◆ The `LongAccumulator`, `LongAdder`, `DoubleAccumulator`, and `DoubleAdder` classes provide better throughput improvements as compared to Atomic variables.
- ◆ The `StampedLock` class implements lock to control read/write access.





- ◆ Parallel computation of streams enables working with streams faster without the risk of threading issues.
- ◆ The new `parallelSort()` method in the `Arrays` class allows parallel sorting of array elements.
- ◆ `RecursiveTask`, similar to `RecursiveAction` extends `ForkJoinTask` to represent tasks that run within a `ForkJoinPool`.

