

# Java 8 CompletableFuture ImageStreamGang Example (Part 3)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)



Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand the design of the Java 8 completable future version of the ImageStreamGang app
- Know how the Java 8 completable future framework is applied to the ImageStreamGang app
- Be aware of the pros & cons of using the completable futures framework

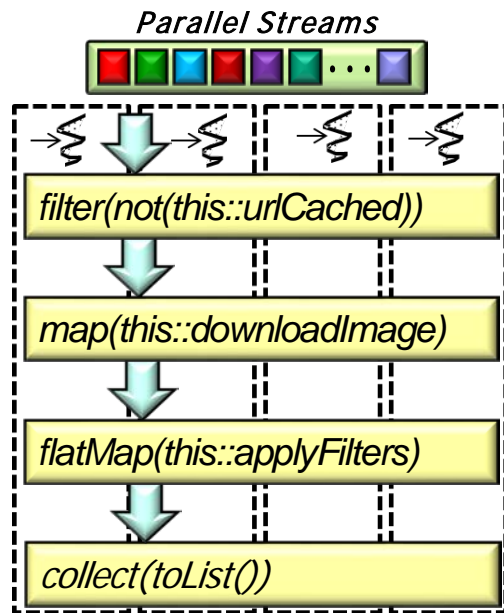
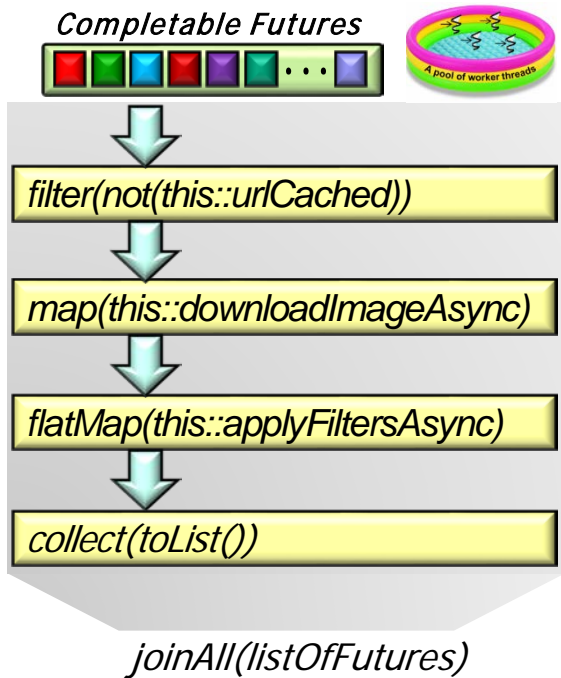


---

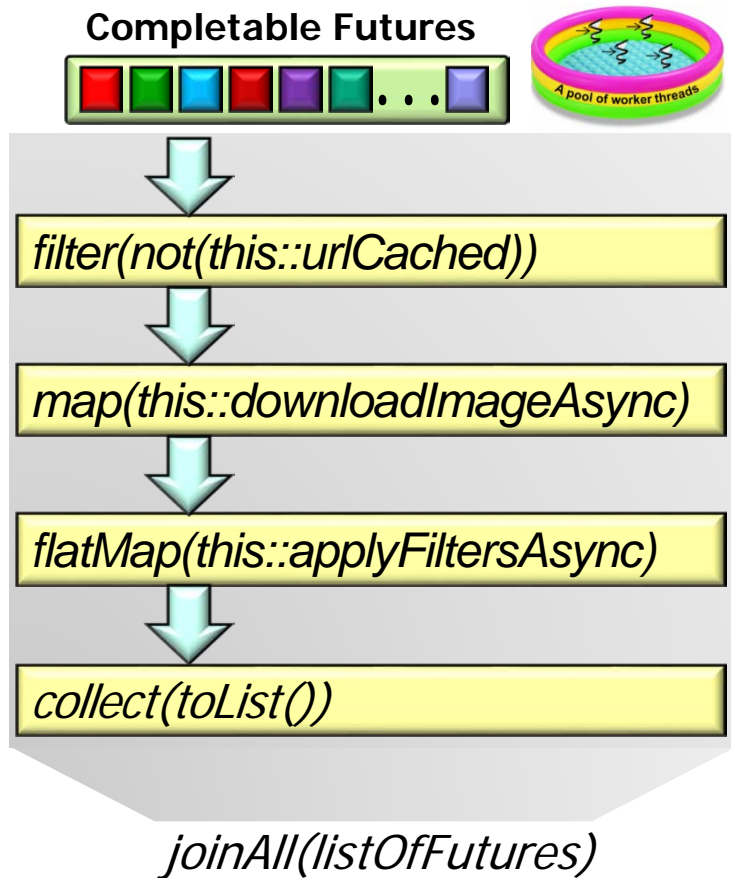
# Evaluating the Completable Futures ImageStreamGang

# Evaluating the Completable Futures ImageStreamGang

- We'll evaluate the Java 8 completable futures framework compared with the parallel streams framework



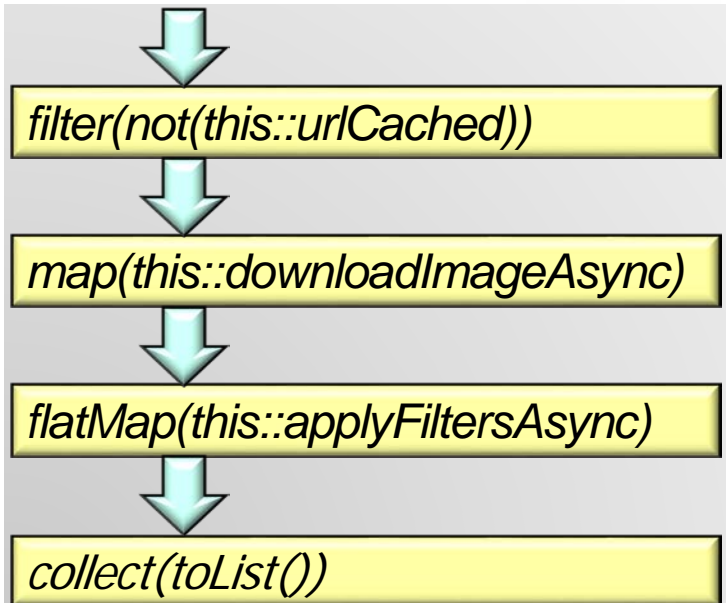
# Evaluating the Completable Futures ImageStreamGang



No explicit synchronization is required in this implementation


# Evaluating the Completable Futures ImageStreamGang

## Completable Futures



*joinAll(listOfFutures)*

Java Language									
java	javac	javadoc	apt	jar	javap	JPDA	JConsole		
Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI	
Deployment			Java Web Start				Java Plug-in		
AWT				Swing			Java 2D		
Accessibility		Drag n Drop		Input Methods		Image I/O	Print Service		Sound
IDL	JDBC		JNDI		RMI		RMI-IIOP		
Beans		Intl Support		Input/Output		JMX	JNI		Math
Networking		Override Mechanism		Security		Serialization	Extension Mechanism		XML JAXP
lang and util		Collections		Concurrency Utilities		JAR	Logging		Management
Preferences API	Ref Objects		Reflection		Regular Expressions		Versioning	Zip	Instrumentation



Java libraries handle any locking needed to read/write to files & connections

# Evaluating the Completable Futures ImageStreamGang

- Java 8 completable futures framework is much more complex to program

```
List<CompletableFuture<List<Image>>>  
  listOfFutures = getInput()  
    .stream()  
    .filter(not(this::urlCached))  
    .map(this::downloadImageAsync)  
    .flatMap(this::applyFiltersAsync)  
    .collect(toList());
```



```
CompletableFuture<List<List<Image>>>  
allImagesDone = StreamsUtils.joinAll(listOfFutures);  
  
int imagesProcessed = allImagesDone.join().stream()  
    .collect(summingInt(List::size));
```



# Evaluating the Completable Futures ImageStreamGang

- Java 8 completable futures framework is much more complex to program

```
List<CompletableFuture<List<Image>>>  
  listOfFutures = getInput()  
    .stream()  
    .filter(not(this::urlCached))  
    .map(this::downloadImageAsync)  
    .flatMap(this::applyFiltersAsync)  
    .collect(toList());
```



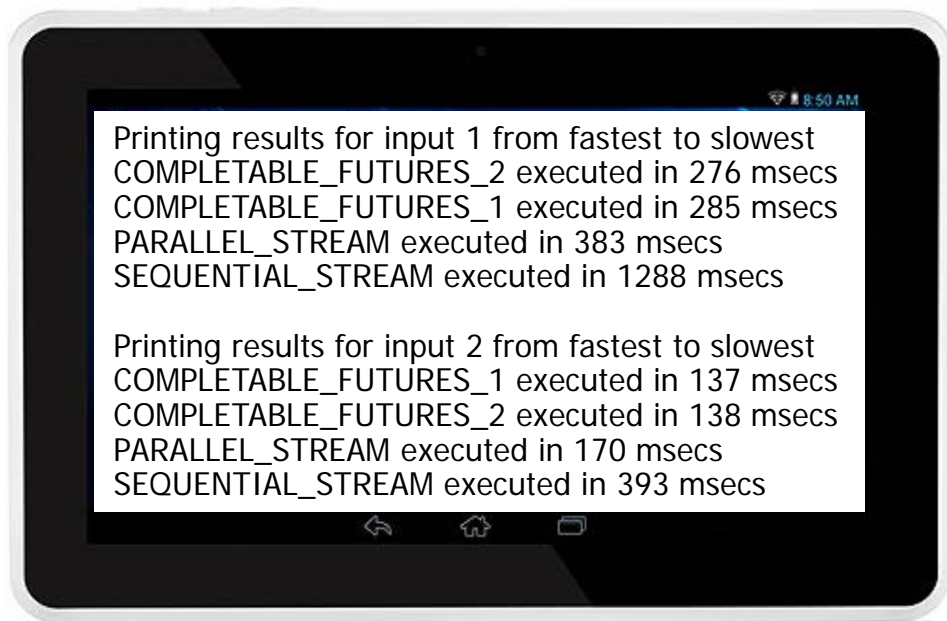
```
CompletableFuture<List<List<Image>>>  
allImagesDone = StreamsUtils.joinAll(listOfFutures);  
  
int imagesProcessed = allImagesDone.join().stream()  
    .collect(summingInt(List::size));
```

In general, asynchrony patterns aren't as well understood by many developers



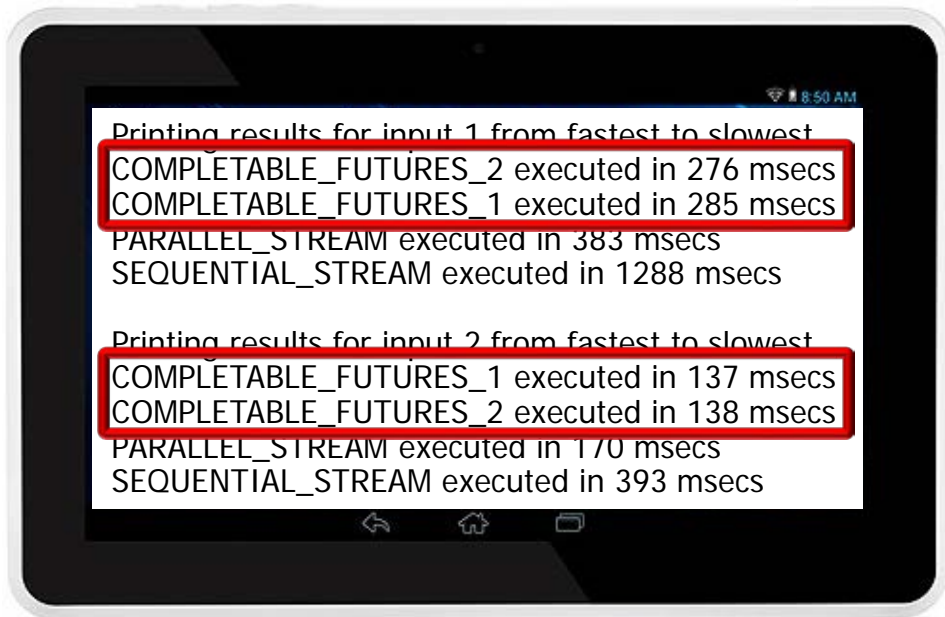
# Evaluating the Completable Futures ImageStreamGang

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks



# Evaluating the Completable Futures ImageStreamGang

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
- Completable futures are more efficient & scalable, but are harder to program



# Evaluating the Completable Futures ImageStreamGang

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
  - Parallel streams are often easier to program, but are less efficient & scalable



# Evaluating the Completable Futures ImageStreamGang

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
  - Parallel streams are often easier to program, but are less efficient & scalable

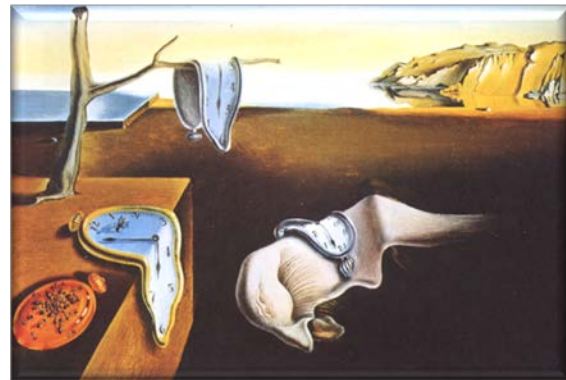


# Evaluating the Completable Futures ImageStreamGang

- Java 9 fixes some completable future limitations

`CompletableFuture`

```
.supplyAsync(  
    () -> findBestPrice("LDN - NYC"),  
    executorService)  
.thenCombine(CompletableFuture  
    .supplyAsync  
        (( ) -> queryExchangeRateFor("GBP")),  
    this::convert)  
.orTimeout(1, TimeUnit.SECONDS)  
.whenComplete((amount, error) -> {  
    if (error == null) { System.out.println("The price is: "  
        + amount + "GBP"); }  
    else { System.out.println("Sorry, no result"); } }));
```



---

# End of Java 8 Completable Futures ImageStreamGang Example (Part 3)