# Overview of Synchrony & Synchronous Operations

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

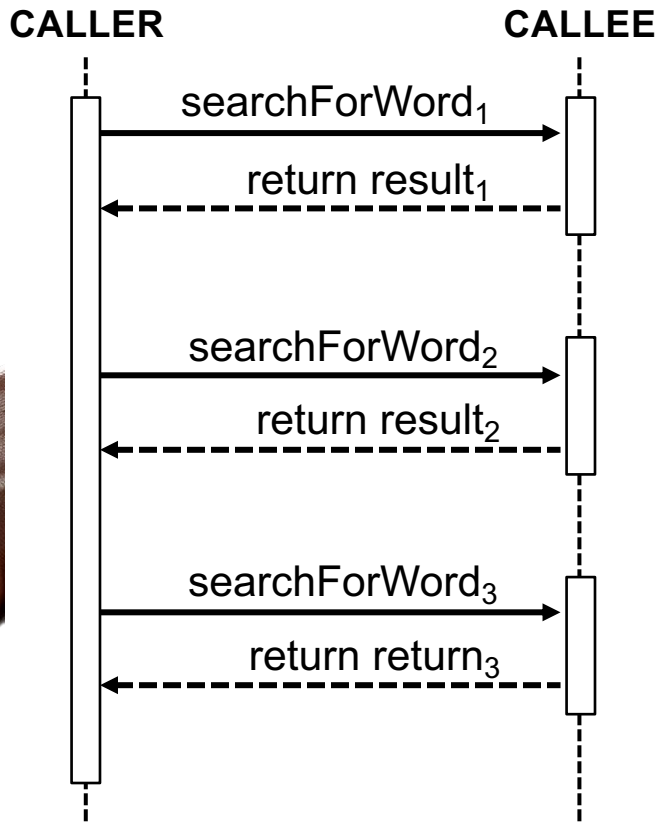
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

# Learning Objectives in this Part of the Lesson

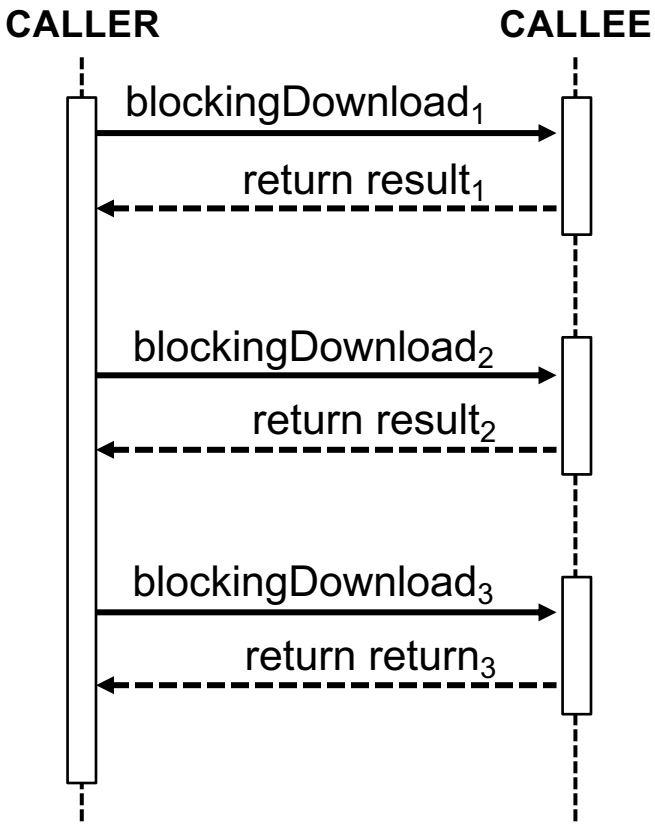
- Understand what synchrony & synchronous operations are



# Overview of Synchrony & Synchronous Operations

# Overview of Synchrony & Synchronous Operations

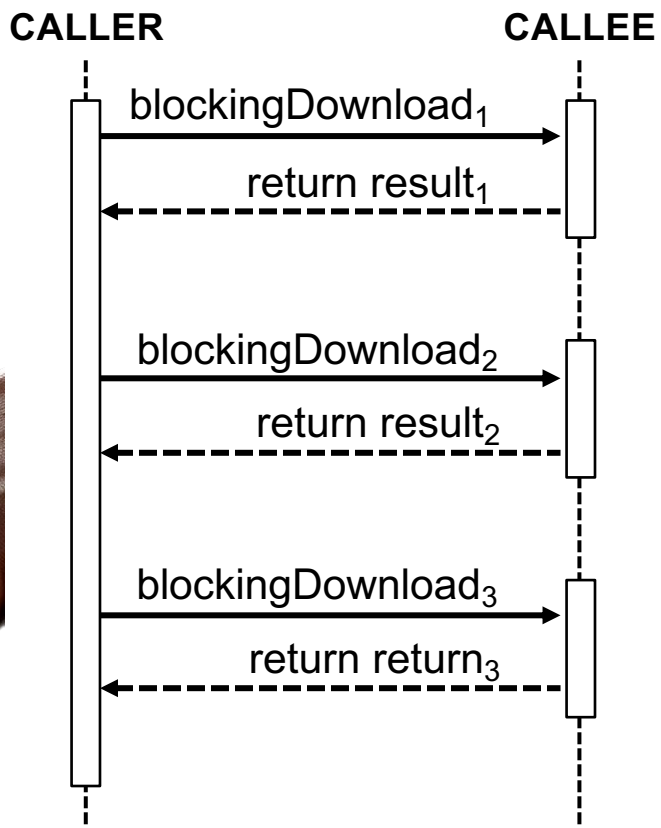
- Method calls in Java programs are largely *synchronous*




e.g., calls on Java collections & behaviors in Java stream aggregate operations

# Overview of Synchrony & Synchronous Operations

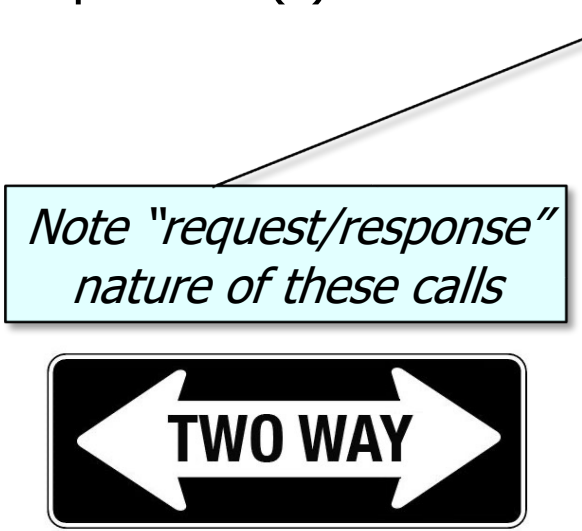
- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish

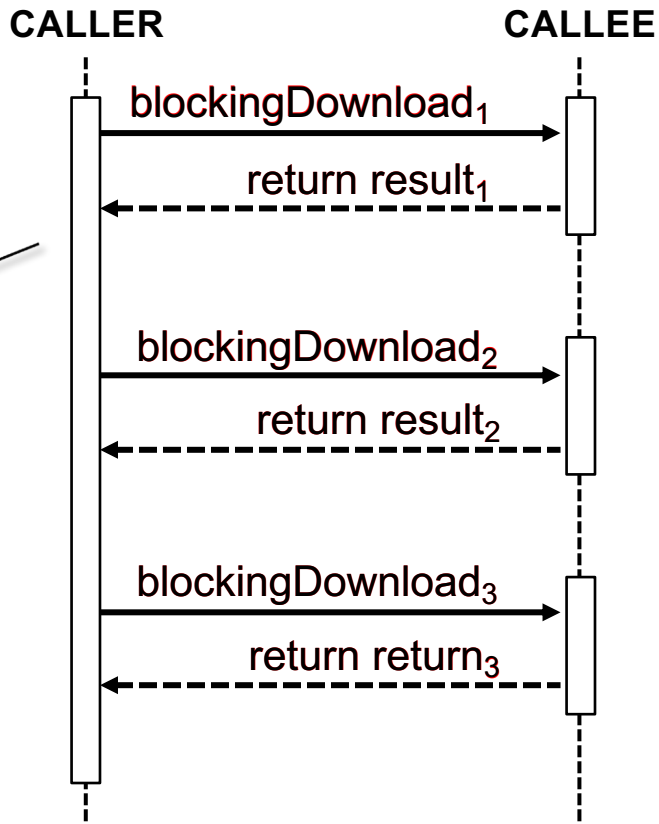



See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

# Overview of Synchrony & Synchronous Operations

- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish




See en.wikipedia.org/wiki/Request-response

# Overview of Synchrony & Synchronous Operations

- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish

*Return an output stream consisting of the images that were downloaded from the URLs in the input stream*

```
void processStream() {
  List<Image> filteredImages =
  getInput()
     .parallelStream()
     .filter(not(this::urlCached))
     .map(this::blockingDownload)
     .mapMulti(this::applyFilters)
     .toList();

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

See "*Implementing Behaviors in the Java Parallel ImageStreamGang Case Study*"

# Overview of Synchrony & Synchronous Operations

- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish

```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .mapMulti(this::applyFilters)
    .toList();

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
    .callInManagedBlock
      (() ->
       downloadImage(url));
}
```

*Transform URL to an Image by downloading each image via its URL*

See livelessons/streams/ImageStreamParallel.java

# Overview of Synchrony & Synchronous Operations

- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish

```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .mapMulti(this::applyFilters)
    .toList();

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlock
     (() ->
      downloadImage(url));
}
```

*"Managed blocker" ensures enough threads in the common fork-join pool*

See lesson on "*The Java Fork-Join Pool: Applying the ManagedBlocker Interface*"

# Overview of Synchrony & Synchronous Operations

- Method calls in Java programs are largely *synchronous*
  - i.e., a callee borrows the thread of its caller until its computation(s) finish

```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .mapMulti(this::applyFilters)
    .toList();

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
    .callInManagedBlock
      (() ->
       downloadImage(url));
}
```

*Synchronously downloads content from URL & converts it into an image*

See livelessons/streams/ImageStreamGang.java

# End of Overview of Synchrony & Synchronous Operations

# Understanding the Pros & Cons of Synchrony


Douglas C. Schmidt

d.schmidt@vanderbilt.edu

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 what synchrony & synchronous operations are
- Motivate the need for Java Future & CompletableFuture mechanisms by understanding the pros & cons of synchrony

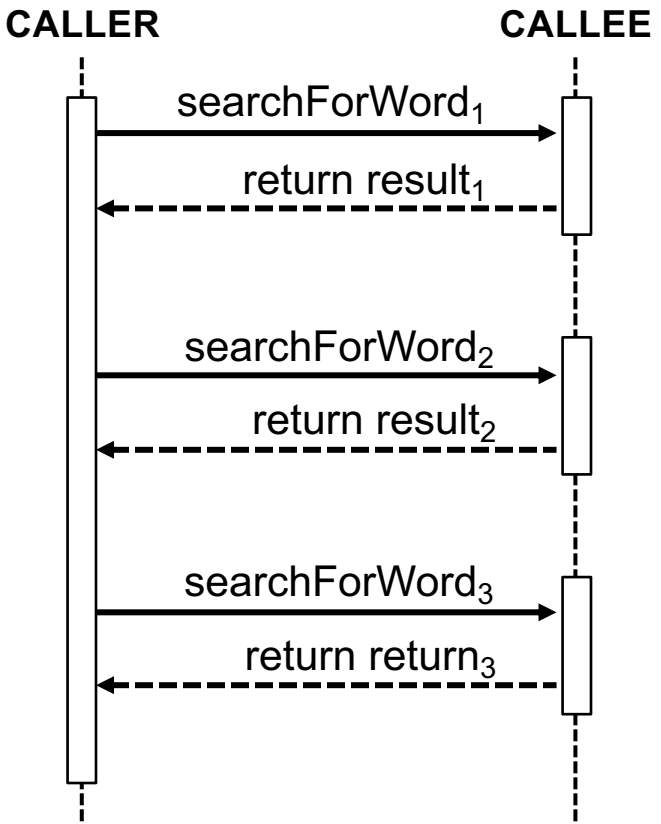
# The Pros of Synchrony

# The Pros of Synchrony

- Pros of synchronous calls

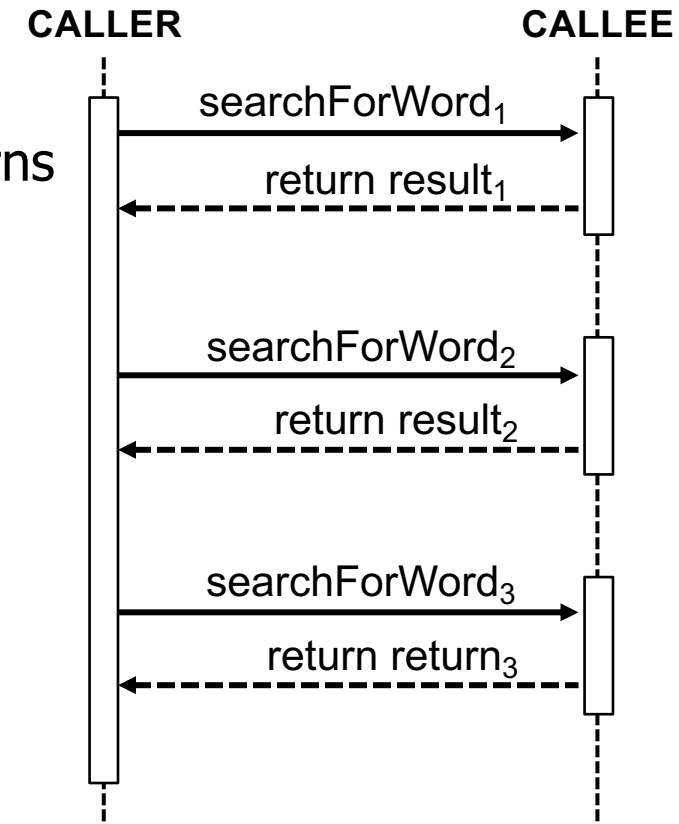
# The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program & debug

# The Pros of Synchrony

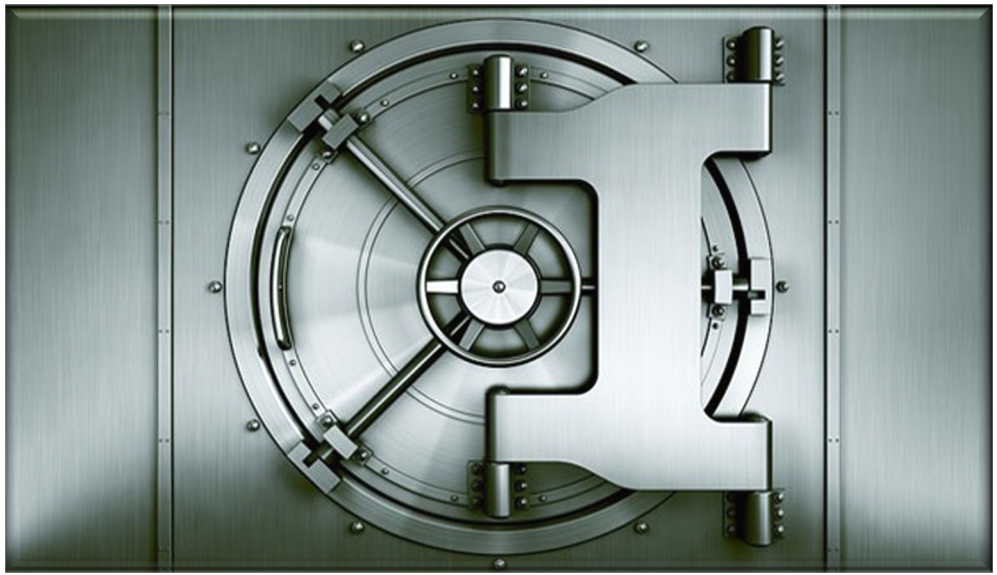
- Pros of synchronous calls
  - "Intuitive" to program & debug, e.g.
    - Maps onto common two-way method patterns

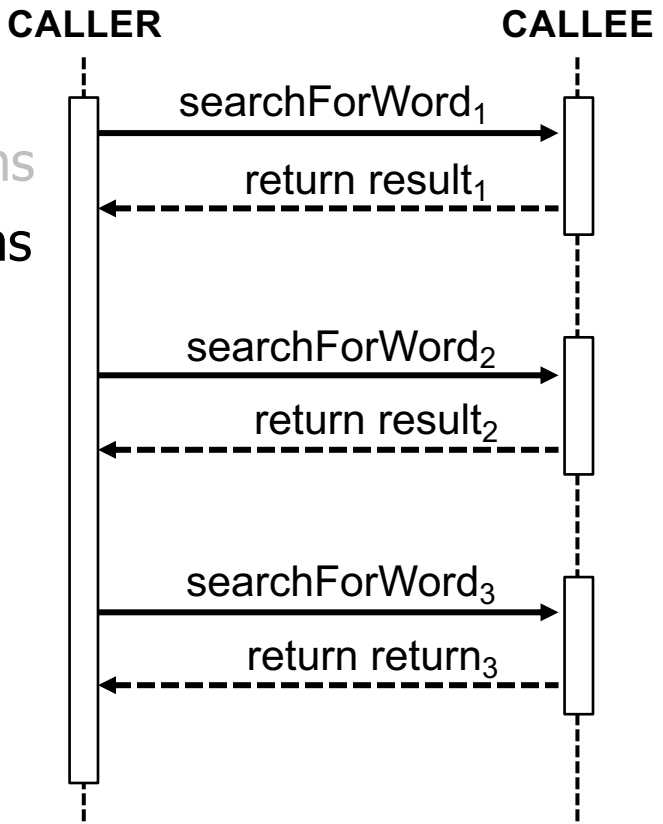




See www.iro.umontreal.ca/~keller/Layla/remote.pdf

# The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program & debug, e.g.
    - Maps onto common two-way method patterns
    - Local caller state retained when callee returns





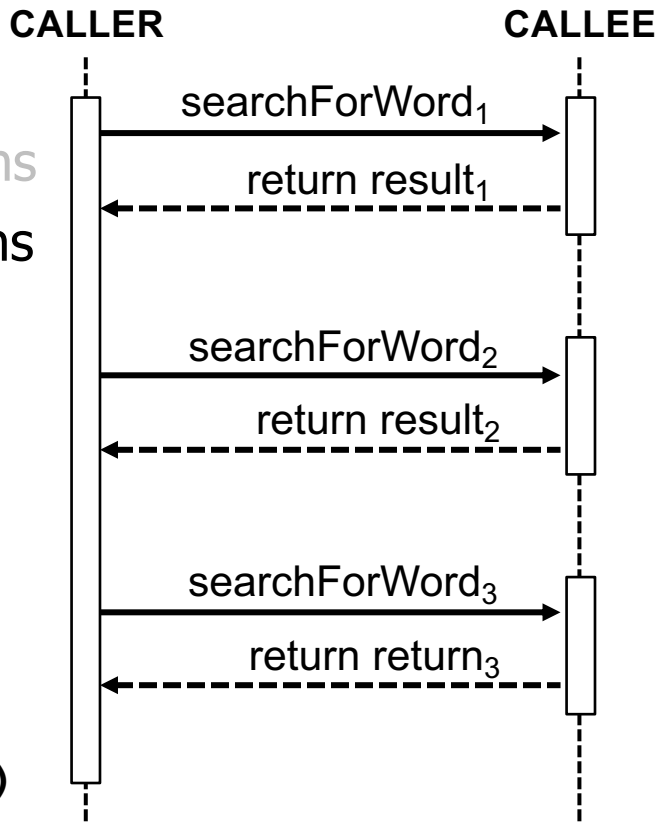

See wiki.c2.com/?ActivationRecord

# The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program & debug, e.g.
    - Maps onto common two-way method patterns
    - Local caller state retained when callee returns

```
byte[] downloadContent(URL url) {
  byte[] buf = new byte[BUFSIZ];
  ByteArrayOutputStream os =
    new ByteArrayOutputStream();

  try(InputStream is = url
      .openStream()) {
    for (int bytes;
         (bytes = is.read(buf)) > 0;)
      os.write(buf, 0, bytes); ...
```




See Java8/ex20/src/main/java/utils/FileAndNetUtils.java

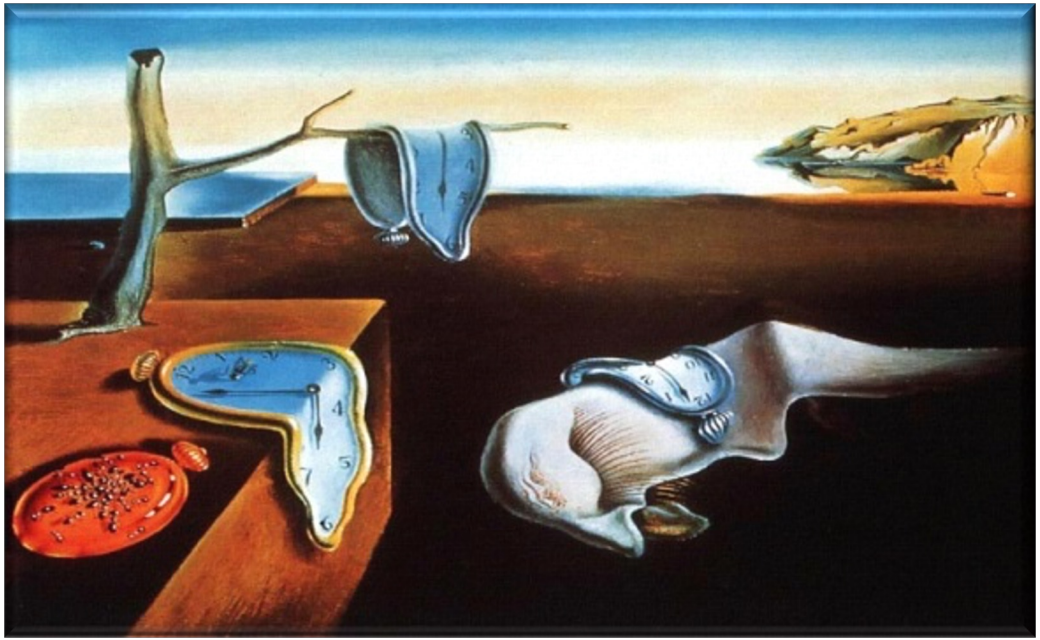
# The Cons of Synchrony

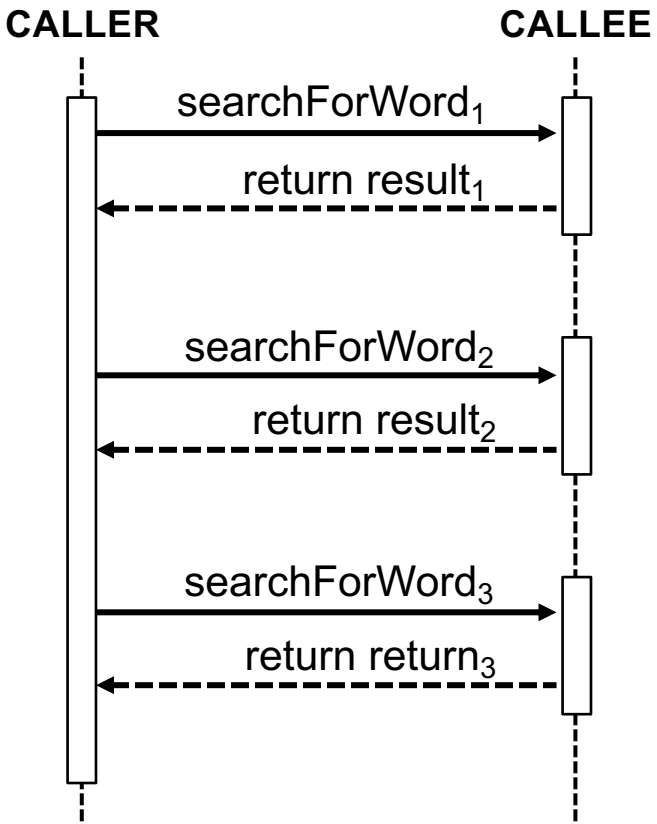
# The Cons of Synchrony

- Cons of synchronous calls

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems






See mincong.io/2020/06/26/completable-future

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
    - Blocking threads incur overhead
      - e.g., synchronization, context switching, data movement, & memory management costs


**Practice**

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
    - Blocking threads incur overhead
    - Selecting right # of threads is hard


*Efficient Performance*

*Efficient Resource Utilization*

```
List<Image> filteredImages = urls
   .parallelStream()
   .filter(not(this::urlCached))
   .map(this::downloadImage)
   .mapMulti(this::applyFilters)
   .toList();
```

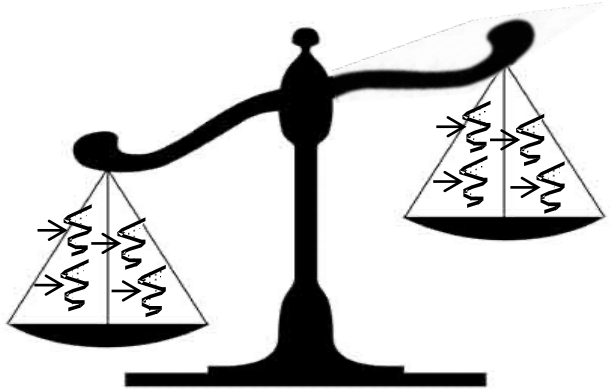
```
Image downloadImage(URL url){
  return new Image
                (url,
                 downloadContent
                   (url));
}
```


See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
    - Blocking threads incur overhead
    - Selecting right # of threads is hard

```
List<Image> filteredImages = urls
   .parallelStream()
   .filter(not(this::urlCached))
   .map(this::downloadImage)
   .mapMulti(this::applyFilters)
   .toList();
```



*Efficient Performance*

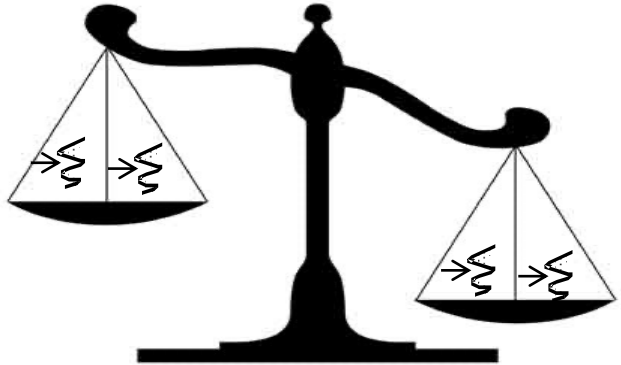
*Efficient Resource Utilization*

*A large # of threads may help to improve performance, but can also waste resources*

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
    - Blocking threads incur overhead
    - Selecting right # of threads is hard

```
List<Image> filteredImages = urls
   .parallelStream()
   .filter(not(this::urlCached))
   .map(this::downloadImage)
   .mapMulti(this::applyFilters)
   .toList();
```



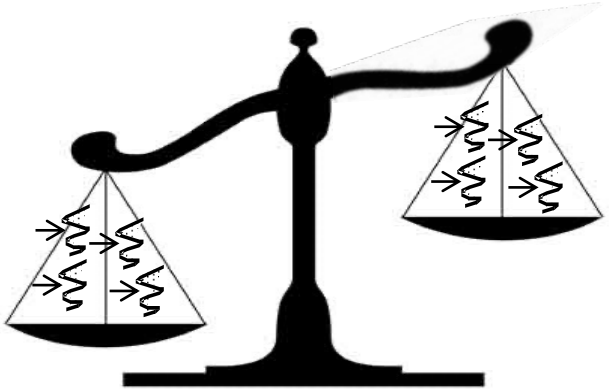
*Efficient Performance*

*Efficient Resource Utilization*

*A small # of threads may conserve resources at the cost of performance*

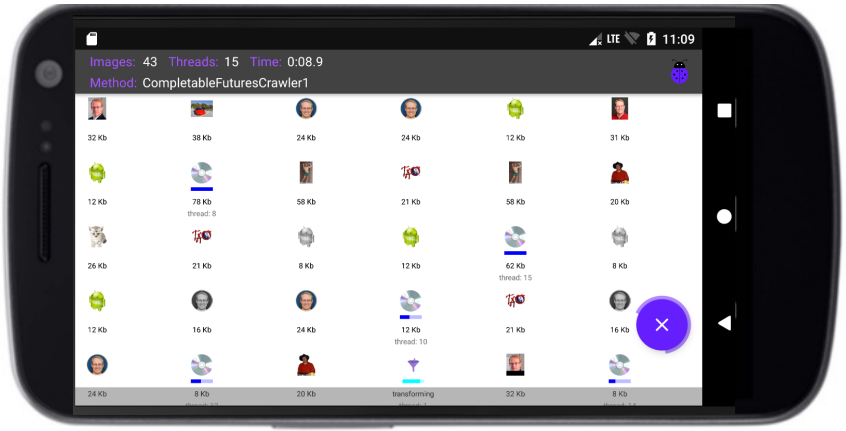
# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
    - Blocking threads incur overhead
    - Selecting right # of threads is hard



*Efficient Performance*

*Efficient Resource Utilization*




*Particularly tricky for I/O-bound programs that need more threads to run efficiently*

# The Cons of Synchrony

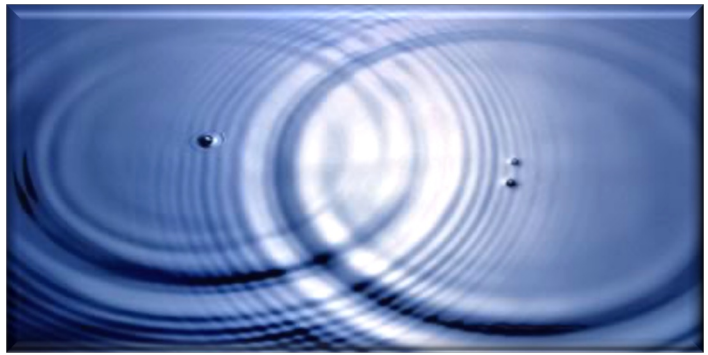
- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool

See lesson on "*The Java Fork-Join Pool: Maximizing Core Utilization w/the Common Fork-Join Pool*"

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool, e.g.
    - Set a system property

```
String desiredThreads = "10";
System.setProperty
  ("java.util.concurrent." +
   "ForkJoinPool.common." +
   "parallelism",
   desiredThreads);
```




It's hard to estimate the total # of threads to set in the common fork-join pool

# The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool, e.g.
    - Set a system property
    - Or use the ManagedBlocker to increase common pool size automatically/temporarily


*ManageBlockers can only be used with the common fork-join pool..*

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html

# End of Understanding the Pros & Cons of Synchrony