Evaluating the ThreadJoinTest Case Study

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

Professor of Computer Science

Institute for Software Integrated Systems

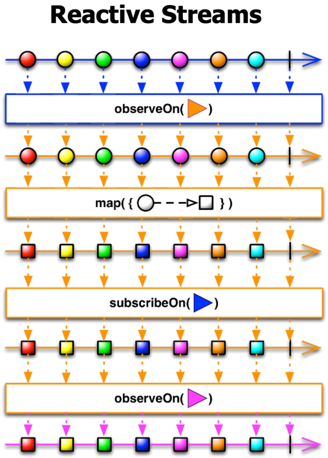
Vanderbilt University
Nashville, Tennessee, USA

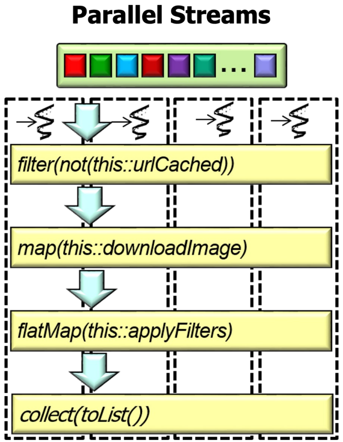
Learning Objectives in this Part of the Lesson

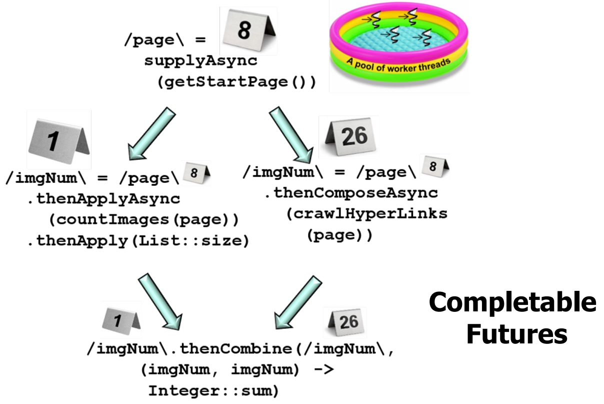
- Understand how Java functional features are applied in an "embarrassingly parallel" program
- Know how to create, start, process, & join Java Thread objects via functional programming features
- Recognize how to use modern Java functional programming features in conjunction with Java Thread methods
- Appreciate the pros & cons of using the Java features in this case study

Learning Objectives in this Part of the Lesson

- Understand how Java functional features are applied in an "embarrassingly parallel" program
- Know how to create, start, process, & join Java Thread objects via functional programming features
- Recognize how to use modern Java functional programming features in conjunction with Java Thread methods
- Appreciate the pros & cons of using the Java features in this case study
  - These "cons" motivate the need for Java's concurrency & parallelism frameworks





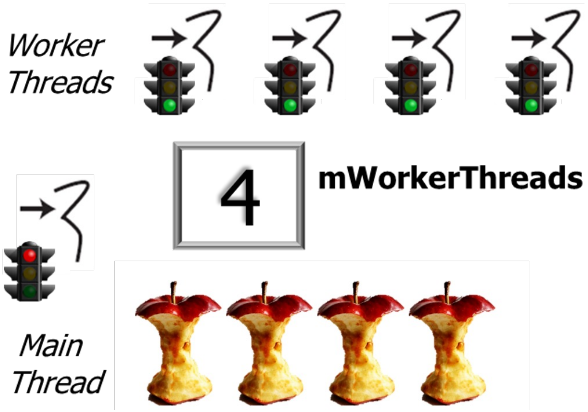


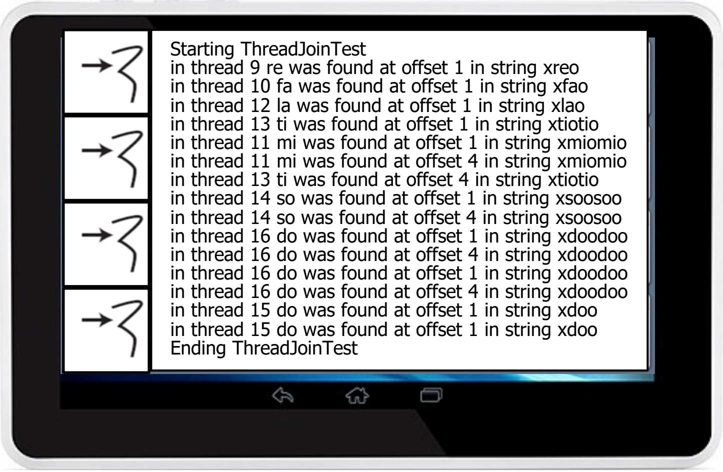
See www.dre.vanderbilt.edu/~schmidt/cs253

Pros of the ThreadJoin Test Program

Pros of the ThreadJoinTest Program

- Foundational Java functional programming features improve the ThreadJoinTest vis-à-vis an earlier Java object-oriented version





See github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/original

Pros of the ThreadJoinTest Program

- The earlier Java object-oriented implementation required more syntax & used traditional for loops

```
for (int i = 0;
     i < mInput.size(); ++i) {
  Thread t = new Thread
    (makeTask(i));

  mWorkerThreads.add(t);
}
...
Runnable makeTask(int i) {
  return new Runnable() {
    public void run() {
      String e = mInput.get(i);
      processInput(e);
    }
  ...
```

See LiveLessons/blob/master/ThreadJoinTest/original/src/ThreadJoinTest.java

Pros of the ThreadJoinTest Program

- The earlier Java object-oriented implementation required more syntax & used traditional for loops

Index-based for loops often suffer from "off-by-one" errors

```
for (int i = 0;
      i < mInput.size(); ++i) {
  Thread t = new Thread
    (makeTask(i));

  mWorkerThreads.add(t);
}
...
Runnable makeTask(int i) {
  return new Runnable() {
    public void run() {
      String e = mInput.get(i);
      processInput(e);
    }
  ...
```

See en.wikipedia.org/wiki/Off-by-one_error

Pros of the ThreadJoinTest Program

- The earlier Java object-oriented implementation required more syntax & used traditional for loops

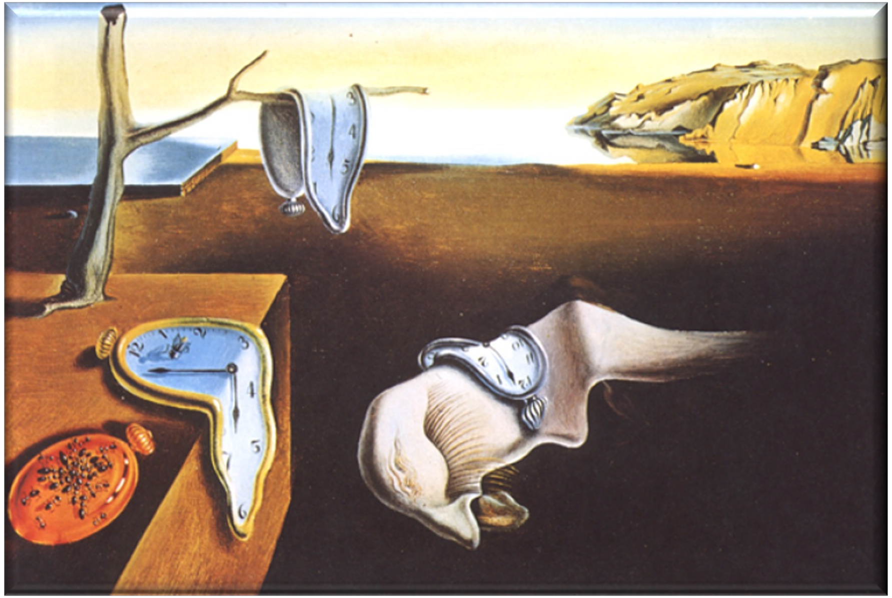
```
for (int i = 0;
     i < mInput.size(); ++i) {
  Thread t = new Thread
    (makeTask(i));

  mWorkerThreads.add(t);
}
...
Runnable makeTask(int i) {
  return new Runnable() {
    public void run() {
      String e = mInput.get(i);
      processInput(e);
    }
  ...
```

Anonymous inner classes are tedious to write..

Pros of the ThreadJoinTest Program

- The earlier Java object-oriented implementation required more syntax & used traditional for loops



```
for (int i = 0;
       i < mInput.size(); ++i) {
   Thread t = new Thread
     (makeTask(i));

   mWorkerThreads.add(t);
}
...
Runnable makeTask(int i) {
   return new Runnable() {
     public void run() {
       String e = mInput.get(i);
       processInput(e);
     }
   ...
```

The object-oriented version was thus more tedious & error-prone to program..

Pros of the ThreadJoinTest Program

- In contrast, the Java functional programming implementation is more concise, extensible, & robust

```
public void run() {
  var workerThreads =
    makeThreads
      (mInputList,
       this::processInput);

  workerThreads
    .forEach(Thread::start);
  ...
```

```
<T, R> List<Thread> makeThreads
   (List<T> inputList,
    Function<T, R> task) {
   ...
   inputList.forEach(input ->
     workerThreads.add
       (new Thread(() -> task.apply(input))));
```

See github.com/douglascraigschmidt/ModernJava/tree/main/CS/ThreadJoinTest

Pros of the ThreadJoinTest Program

- In contrast, the Java functional programming implementation is more concise, extensible, & robust
  - e.g., Java features like forEach(), functional interfaces, method references, & lambda expressions

```
<T, R> List<Thread> makeThreads
  (List<T> inputList,
   Function<T, R> task) {
  ...
  inputList.forEach(input ->
    workerThreads.add
      (new Thread(() -> task.apply(input))));
```

```
public void run() {
  var workerThreads =
    makeThreads
      (mInputList,
       this::processInput);

  workerThreads
    .forEach(Thread::start);
  ...
```

Pros of the ThreadJoinTest Program

- In contrast, the Java functional programming implementation is more concise, extensible, & robust
  - e.g., Java features like forEach(), functional interfaces, method references, & lambda expressions

```
public void run() {
  var workerThreads =
    makeThreads
      (mInputList,
       this::processInput);

  workerThreads
    .forEach(Thread::start);
  ...
```

```
<T, R> List<Thread> makeThreads
  (List<T> inputList,
   Function<T, R> task){
  ...
  inputList.forEach(input ->
    workerThreads.add
      (new Thread(() -> task.apply(input)));
```

The forEach() method avoids "off-by-one" fence-post errors

See en.wikipedia.org/wiki/Off-by-one_error

Pros of the ThreadJoinTest Program

- In contrast, the Java functional programming implementation is more concise, extensible, & robust

```
public void run() {
  var workerThreads =
    makeThreads
      (this::processInput);

  workerThreads
    .forEach(Thread::start);
  ...
```

```
<T, R> List<Thread> makeThreads
  (List<T> inputList,
   Function<T, R> task){
  ...
  inputList.forEach(input ->
    workerThreads.add
      (new Thread(() -> task.apply(input))));
```

Functional interfaces, method references, & lambda expressions simplify behavior parameterization

See blog.indrek.io/articles/java-8-behavior-parameterization

Cons of the ThreadJoin Test Program

Cons of the ThreadJoinTest Program

- There are limitations with foundational Java functional programming features

These features are not all rainbows & unicorns!!

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version

Accidental complexities arise from limitations with software techniques, tools, & methods

See en.wikipedia.org/wiki/No_Silver_Bullet

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects

```
public void run() {
  var workerThreads =
    makeThreads
      (this::processInput);

  workerThreads
    .forEach(Thread::start);

  workerThreads
    .forEach(thread -> {
      try { thread.join(); }
      catch(Exception e) {
        throw new
         RuntimeException(e);
      }}); ...
```

You must remember to start each Thread!

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects


```
public void run() {
  var workerThreads =
    makeThreads
      (this::processInput);

  workerThreads
    .forEach(Thread::start);

  workerThreads
    .forEach(thread -> {
      try { thread.join(); }
      catch(Exception e) {
        throw new
         RuntimeException(e);
      }}); ...
```

Note the verbosity of handling checked exceptions in modern Java programs..

See codingjunkie.net/functional-iterface-exceptions

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects

```
public void run() {
  var workerThreads =
    makeThreads
      (this::processInput);

  workerThreads
    .forEach(Thread::start);

  workerThreads
    .forEach(rethrowConsumer
               (Thread::join));
```

A helper class can enable less verbose use of checked exceptions in Java functional programs, though there is some controversy about this type of "exception laundering"

See stackoverflow.com/a/27644392/3312330

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects
  - One concurrency model supported
    - "thread-per-work" hard-codes the # of threads to # of input strings

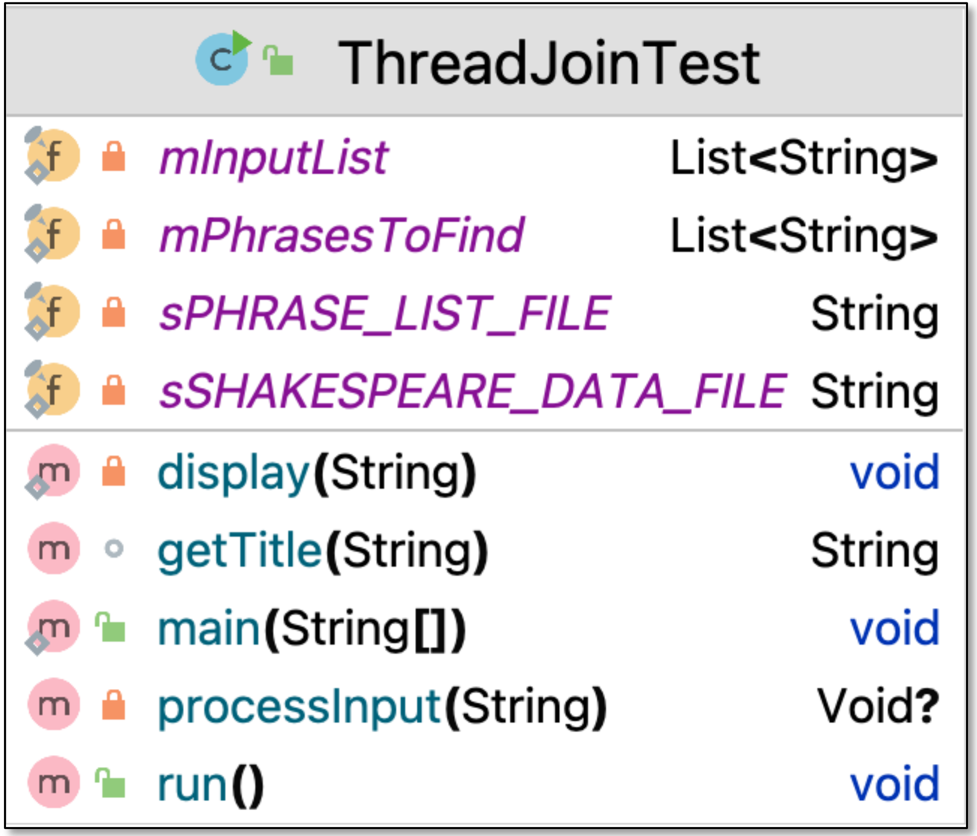
```
<T, R> List<Thread> makeThreads
   (List<T> inputList,
    Function<T, T> task){
  List<Thread> workerThreads =
    new ArrayList<>();

  inputList.forEach(input ->
    workerThreads.add
      (new Thread(()
        -> task.apply(input))));

  return workerThreads;
}
```

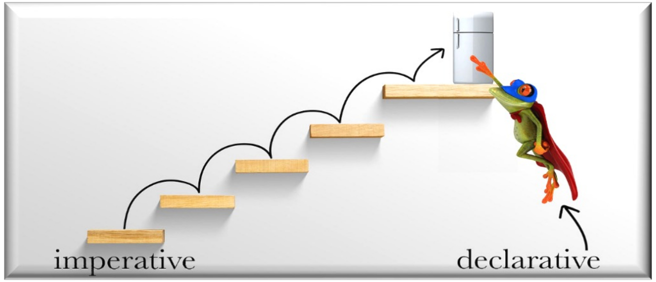
Cons of the ThreadJoinTest Program

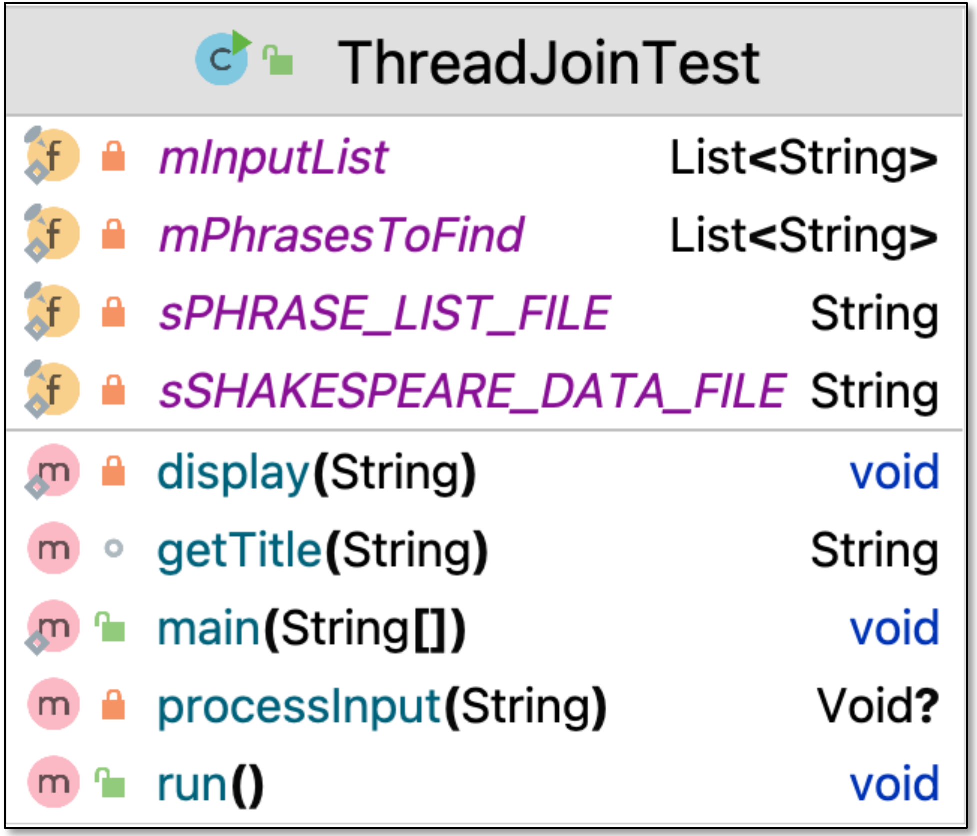
- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects
  - One concurrency model supported
  - Not easily extensible without major changes to the code



Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects
  - One concurrency model supported
  - Not easily extensible without major changes to the code

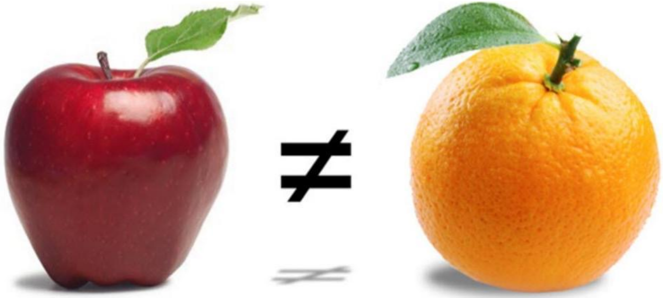




The ThreadJoinTest implementation is insufficiently declarative!

Cons of the ThreadJoinTest Program

- "Accidental complexity" still lurks in the functional programming version, e.g.
  - Manually creating, starting, & joining Thread objects
  - One concurrency model supported
  - Not easily extensible without major changes to the code



```
var workerThreads = makeThreads
                      (this::processInput);

workerThreads
   .forEach(Thread::start);

workerThreads
   .forEach(rethrowConsumer
      (Thread::join));
```

Concurrent implementation vs. sequential implementation

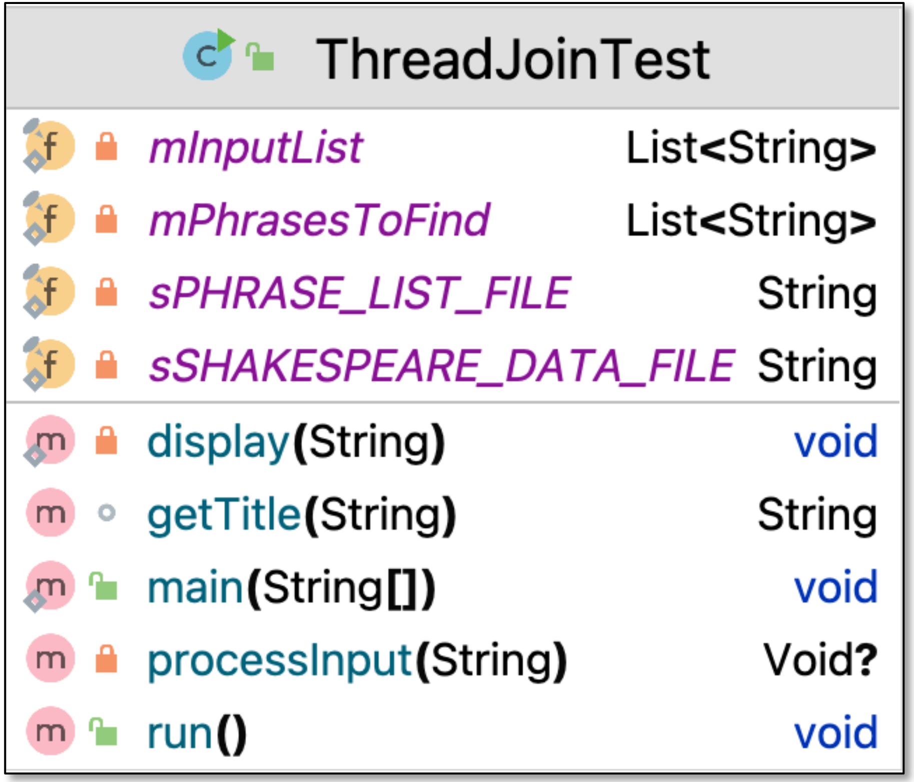
```
mInputList
   .forEach(this::processInput);
```

The structure of the concurrent code is much different than the sequential code

Addressing the Cons of the ThreadJoinTest Program

Addressing the Cons of the ThreadJoinTest Program

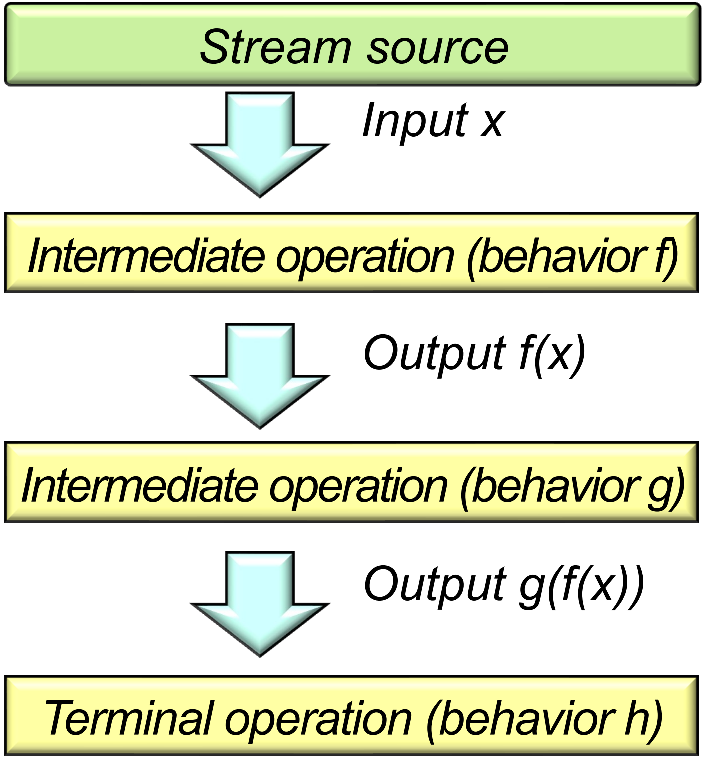
- Solutions require more than foundational Java functional programming features



See www.youtube.com/watch?v=1OpAgZvYXLQ

Addressing the Cons of the ThreadJoinTest Program

- Solutions require more than foundational Java functional programming features

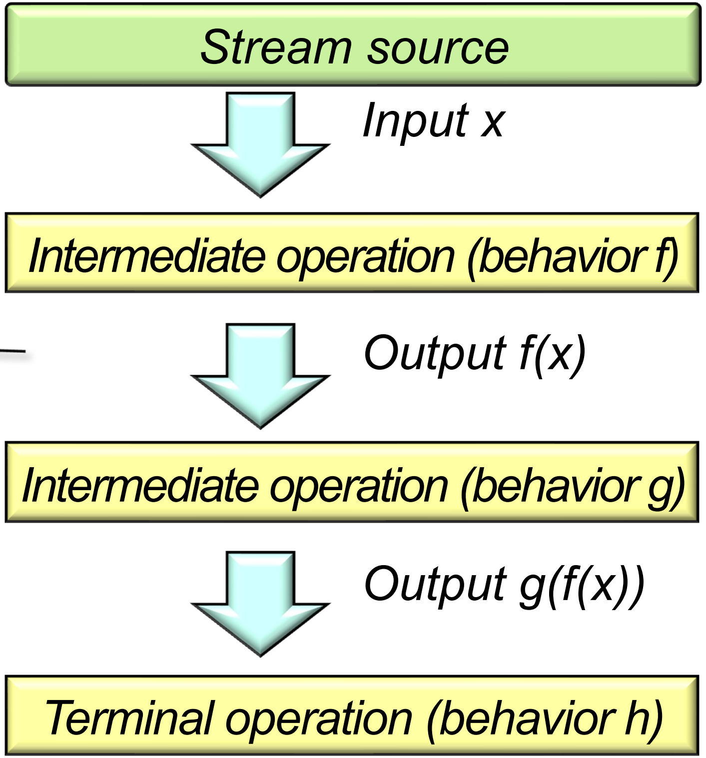


See docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Addressing the Cons of the ThreadJoinTest Program

- Solutions require more than foundational Java functional programming features

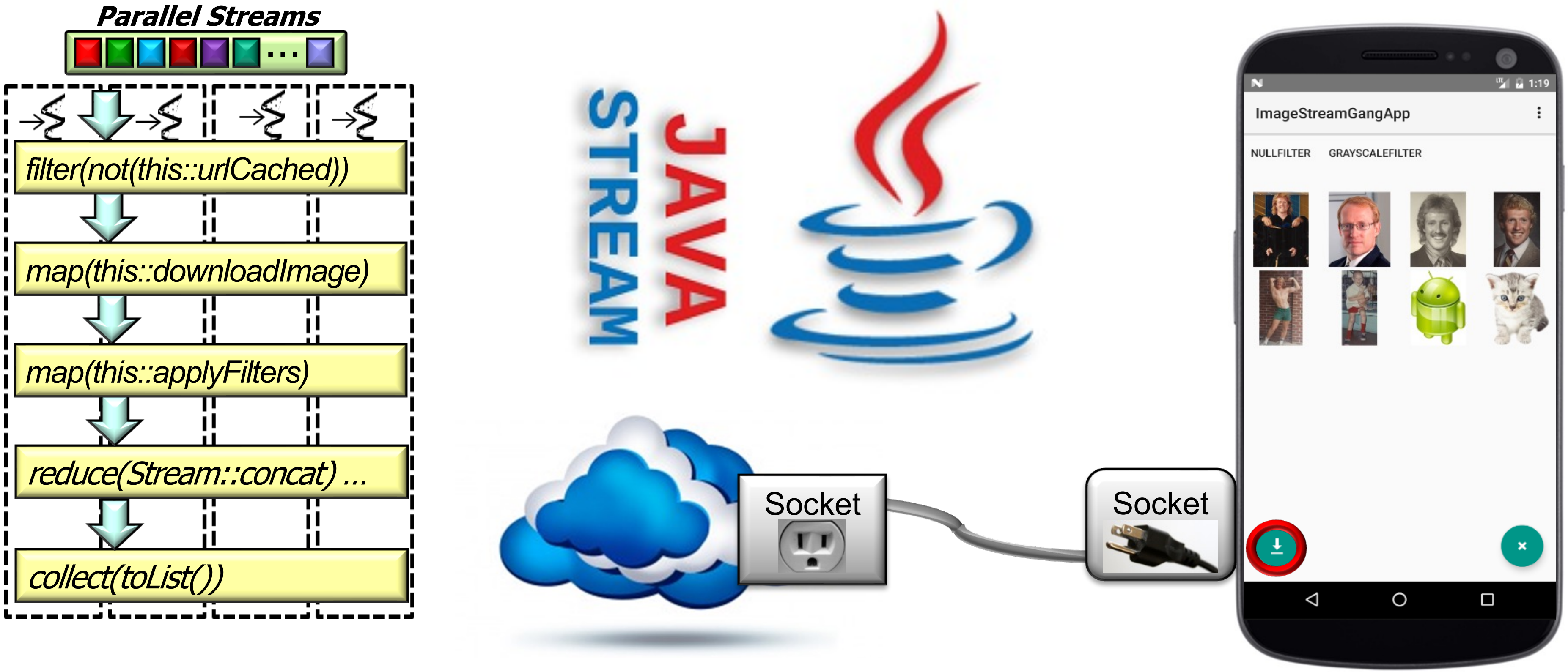
Java Streams support functional-style operations on sequences of elements, such as map-reduce transformations, filtering, slicing, searching, matching, etc.



See www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html

Addressing the Cons of the ThreadJoinTest Program

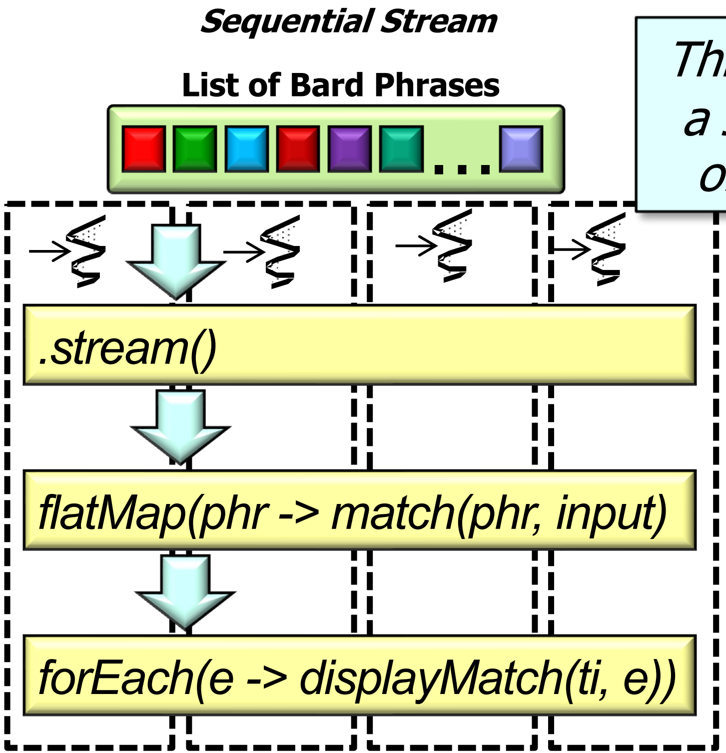
- Solutions require more than foundational Java functional programming features



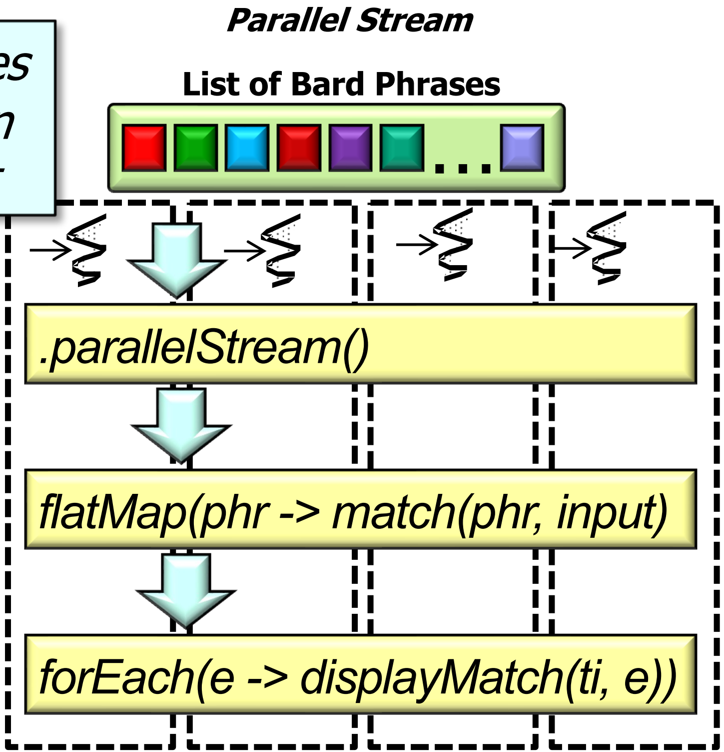
See www.dre.vanderbilt.edu/~schmidt/cs253

Addressing the Cons of the ThreadJoinTest Program

- Solutions require more than foundational Java functional programming features



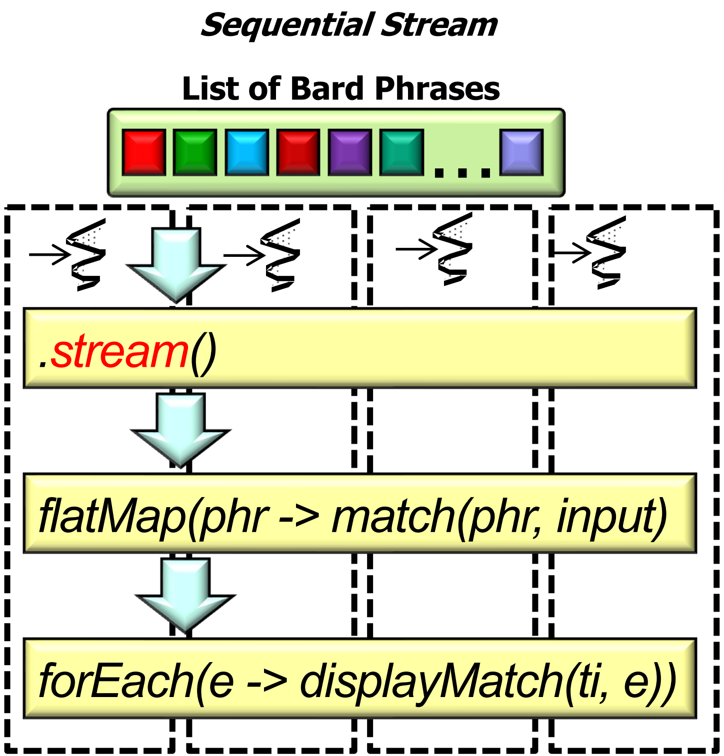
This case study provides a Java Streams version of the ThreadJoinTest



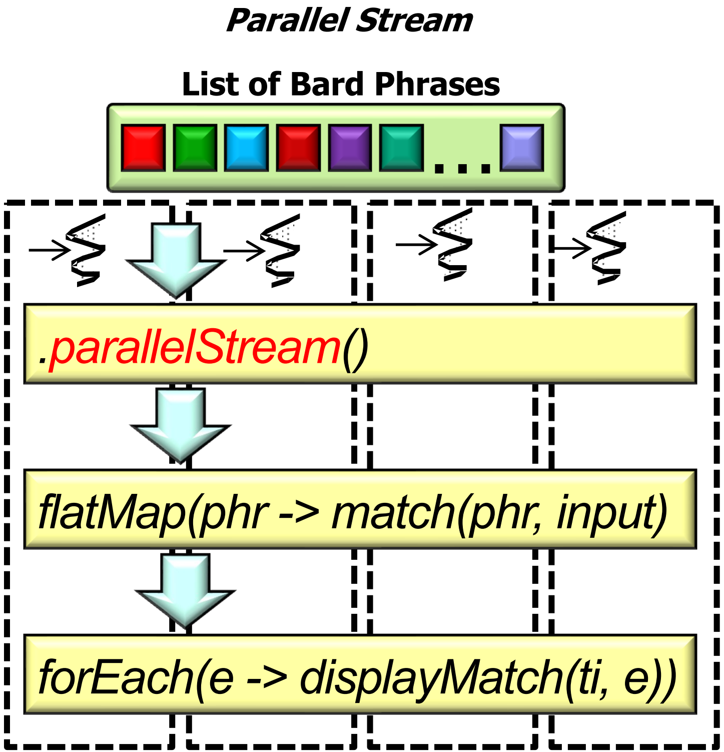
See github.com/douglascraigschmidt/ModernJava/tree/main/CS/BardStreamTest

Addressing the Cons of the ThreadJoinTest Program

- Solutions require more than foundational Java functional programming features



VS



The structure of the sequential code is nearly identical to the concurrent code

End of Evaluating the ThreadJoinTest Case Study