Applying Java Functional Programming
Features: Evaluating Pros & Cons

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 Lesson

- Understand how Java functional programming features are applied in a simple parallel program
- Know how to start & join Java threads via functional programming features
- Appreciate the pros & cons of using Java features in this example

These “cons” motivate the need for Java function parallelism frameworks
Pros of the ThreadJoinTest Program
Pros of the ThreadJoinTest Program

• Foundational Java FP features improve the program vis-à-vis original OO Java version

See github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/original
Pros of the ThreadJoinTest Program

• Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.

• The OO Java version has more syntax & traditional for loops

```java
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);
        }
    }
    ...
```
Pros of the ThreadJoinTest Program

- Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
  - The OO Java version has more syntax & traditional for loops

```java
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);
        }
    }
...  
Index-based for loops often suffer from "off-by-one" errors
```

Pros of the ThreadJoinTest Program

- Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
  - The OO Java version has more syntax & traditional for loops

```java
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

• Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
• The OO Java version has more syntax & traditional for loops

```java
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 OO Java version is thus more tedious & error-prone to program..
Pros of the ThreadJoinTest Program

- Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
  - The OO Java version has more syntax & traditional for loops
  - The FP Java implementation is more concise, extensible, & robust

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (this::processInput);

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

List<Thread> makeWorkerThreads
    (Function<String, Void> task) {
    ...
    mInputList.forEach(input ->
        workerThreads.add
            (new Thread(() -> task.apply(input))));
}

E.g., declarative Java features such as `forEach()`, functional interfaces, method references, & lambda expressions
Pros of the ThreadJoinTest Program

- Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
  - The OO Java version has more syntax & traditional for loops
- The FP Java implementation is more concise, extensible, & robust

```
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
        (this::processInput);

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

    List<Thread> makeWorkerThreads
        (Function<String, Void> task) {
        ...
        mInputList.forEach(input ->
            workerThreads.add
                (new Thread(() -> task.apply(input))));
```

The forEach() method avoids "off-by-one" fence-post errors
Pros of the ThreadJoinTest Program

- Foundational Java FP features improve the program vis-à-vis original OO Java version, e.g.
  - The OO Java version has more syntax & traditional for loops
  - The FP Java implementation is more concise, extensible, & robust

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (this::processInput);
    workerThreads
        .forEach(Thread::start);
    ...
}

List<Thread> makeWorkerThreads
    (Function<String, Void> task) {
    ...
    mInputList.forEach(input ->
        workerThreads.add
            (new Thread(() -> task.apply(input))));
}
```

Functional interfaces, method references, & lambda expressions simplify behavioral parameterization
Cons of the ThreadJoinTest Program
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java FP version

See en.wikipedia.org/wiki/No_Silver_Bullet

Accidental complexities arise from limitations with software techniques, tools, & methods
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java FP version, e.g.
- Manually creating, starting, & joining threads

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (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

- There’s still “accidental complexity” in the Java FP version, e.g.
- Manually creating, starting, & joining threads

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (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 Java 8 programs.*

See [codingjunkie.net/functional-interface-exceptions](http://codingjunkie.net/functional-interface-exceptions)
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java FP version, e.g.
  - Manually creating, starting, & joining threads

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (this::processInput);

    workerThreads
        .forEach(Thread::start);

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

A helper class enables less verbosely use of checked exceptions in Java FP programs

See stackoverflow.com/a/27644392/3312330
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java FP version, e.g.
  - Manually creating, starting, & joining threads
- Only one parallelism model supported
  - “thread-per-work” hard-codes the # of threads to # of input strings

```java
List<Thread> makeWorkerThreads(Function<String, Void> task) {
    List<Thread> workerThreads = new ArrayList<>();
    
    mInputList.forEach(input -> {
        workerThreads.add(new Thread(() -> task.apply(input)));
    });

    return workerThreads;
}
```
Cons of the ThreadJoinTest Program

- There's still “accidental complexity” in the Java FP version, e.g.
  - Manually creating, starting, & joining threads
  - Only one parallelism model supported
  - Not easily extensible without major changes to the code
  - e.g., insufficiently declarative
Cons of the ThreadJoinTest Program

- Solving these problems requires more than the foundational Java FP features

See [www.dre.vanderbilt.edu/~schmidt/DigitalLearning](http://www.dre.vanderbilt.edu/~schmidt/DigitalLearning)
Cons of the ThreadJoinTest Program

- Solving these problems requires more than the foundational Java FP features

Java 8’s parallelism frameworks provide an FP façade around its earlier OO features

See en.wikipedia.org/wiki/Facade_pattern
End of Applying Java Functional Programming Features: Evaluating Pros & Cons