Learning Objectives in this Lesson

• Understand key aspects of functional programming
Learning Objectives in this Lesson

- Understand key aspects of functional programming
- Contrasted with object-oriented programming

We’ll show some Java 8 code fragments that will be covered in more detail later.
Learning Objectives in this Lesson

• Understand key aspects of functional programming

• Recognize the benefits of applying functional programming in Java 8
Learning Objectives in this Lesson

• Understand key aspects of functional programming

• Recognize the benefits of applying functional programming in Java 8

• Especially when used in conjunction with object-oriented programming

Again, we’ll show Java 8 code fragments that’ll be covered in more detail later
Overview of Programming Paradigms in Java 8
Overview of Programming Paradigms in Java 8

- Java 8 is a “hybrid” that combines the object-oriented & functional paradigms
Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm

See [en.wikipedia.org/wiki/Imperative_programming](en.wikipedia.org/wiki/Imperative_programming)
Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm
  - e.g., a program consists of commands for the computer to perform

> Imperative programming focuses on describing how a program operates via statements that change its state

![Diagram showing relationships between programming paradigms](image-url)
Object-oriented programming is an “imperative” paradigm

- A program consists of commands for the computer to perform

```java
List<String> zap(List<String> lines, String omit) {
    List<String> res = new ArrayList<>();
    for (String line : lines)
        if (!omit.equals(line))
            res.add(line);
    return res;
}
```

Languages:
- C++, Java, C#
- C, FORTRAN

Imperatively remove a designated string from a list of strings
Overview of Programming Paradigms in Java 8

- Object-oriented programming is an “imperative” paradigm
- e.g., a program consists of commands for the computer to perform

```java
List<String> zap(List<String> lines, String omit) {
    List<String> res = new ArrayList<>();
    for (String line : lines)
        if (!omit.equals(line))
            res.add(line);
    return res;
}
```

- e.g., C++, Java, C#
- e.g., C, FORTRAN

This inherently sequential code applies the Accumulator anti-pattern

Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm

See [en.wikipedia.org/wiki/Declarative_programming](en.wikipedia.org/wiki/Declarative_programming)
Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
- E.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

*Declarative programming focuses on “what” computations to perform, not “how” to compute them*
Overview of Programming Paradigms in Java 8

• Conversely, functional programming is a “declarative” paradigm
• e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```java
List<String> zap(List<String> lines, String omit) {
    return lines
        .stream()
        .filter(not(omit::equals))
        .collect(toList());
}
```

e.g., Prolog, e.g., ML, Haskell

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex0](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex0)
Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a "declarative" paradigm
- e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```java
List<String> zap(List<String> lines, String omit) {
    return lines.stream()
        .filter(not(omit::equals))
        .collect(toList());
}
```

- e.g., Prolog, e.g., ML, Haskell

Note "fluent" programming style with cascading method calls

See [en.wikipedia.org/wiki/Fluent_interface](en.wikipedia.org/wiki/Fluent_interface)
Overview of Programming Paradigms in Java 8

- Conversely, functional programming is a “declarative” paradigm
- e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```java
List<String> zap(List<String> lines, String omit) {
    return lines
        .parallelStream()
        .filter(not(omit::equals))
        .collect(toList());
}
```

See [docs.oracle.com/javase/tutorial/collections/streams/parallelism.html](http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html)
Overview of Programming Paradigms in Java 8

• Conversely, functional programming is a “declarative” paradigm
• e.g., a program expresses computational logic without describing control flow or explicit algorithmic steps

```java
List<String> zap(List<String> lines, String omit) {
    return lines
        .parallelStream()
        .filter(not(omit::equals))
        .collect(toList());
}
```

Code was parallelized with minuscule changes since it’s declarative & stateless!
Overview of Functional Programming in Java 8
Overview of Functional Programming in Java 8

• Functional programming has its roots in lambda calculus

See [en.wikipedia.org/wiki/Functional_programming](en.wikipedia.org/wiki/Functional_programming)
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
- Computations are treated as the evaluation of mathematical functions

See [en.wikipedia.org/wiki/Functional_programming#Pure_functions](en.wikipedia.org/wiki/Functional_programming#Pure_functions)

Note “function composition”: the output of one function serves as the input to the next function, etc.
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,

  Computations are treated as the evaluation of mathematical functions

\[
\text{long factorial(long n) \{ return LongStream} \\
\quad .rangeClosed(1, n)} \\
\quad .parallel()} \\
\quad .reduce(1, (a, b) \rightarrow a * b); \\
\}
\]

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
  - Computations are treated as the evaluation of mathematical functions
  - Changing state & mutable data are discouraged to avoid various hazards

See en.wikipedia.org/wiki/Side_effect_(computer_science)
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
  - Computations are treated as the evaluation of mathematical functions
  - Changing state & mutable data are discouraged to avoid various hazards

```java
class Total {
    public long mTotal = 1;
    public void mult(long n) {
        mTotal *= n;
    }
}
```

```java
long factorial(long n) {
    Total t = new Total();
    LongStream.rangeClosed(1, n)
        .parallel()
        .forEach(t::mult);
    return t.mTotal;
}
```

- Shared mutable state

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
- Computations are treated as the evaluation of mathematical functions
- Changing state & mutable data are discouraged to avoid various hazards

```java
long factorial(long n) {
    Total t = new Total();
    LongStream.rangeClosed(1, n)
        .parallel()
        .forEach(t::mult);
    return t.mTotal;
}
```

```java
class Total {
    public long mTotal = 1;
    public void mult(long n) {
        mTotal *= n;
    }
}
```

Beware of race conditions!!!

See [en.wikipedia.org/wiki/Race_condition#Software](en.wikipedia.org/wiki/Race_condition#Software)
Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
- Computations are treated as the evaluation of mathematical functions
- Changing state & mutable data are discouraged to avoid various hazards

```java
long factorial(long n) {
    Total t = new Total();
    LongStream.rangeClosed(1, n)
        .parallel()
        .forEach(t::mult);
    return t.mTotal;
}
```

```java
class Total {
    public long mTotal = 1;
    public void mult(long n) {
        mTotal *= n;
    }
}
```

Beware of inconsistent memory visibility

Overview of Functional Programming in Java 8

- Functional programming has its roots in lambda calculus, e.g.,
  - Computations are treated as the evaluation of mathematical functions
  - Changing state & mutable data are discouraged to avoid various hazards

```java
long factorial(long n) {
    Total t = new Total();
    LongStream.rangeClosed(1, n)
        .parallel()
        .forEach(t::mult);
    return t.mTotal;
}

class Total {
    public long mTotal = 1;
    public void mult(long n) {
        mTotal *= n; }
}
```

In Java you must avoid these hazards, i.e., the compiler & JVM won't save you.

Only you can prevent concurrency hazards!
Overview of Functional Programming in Java 8

• Functional programming has its roots in lambda calculus, e.g.,
  • Computations are treated as the evaluation of mathematical functions
  • Changing state & mutable data are discouraged to avoid various hazards
  • Instead, the focus is on “immutable” objects
    • i.e., objects whose state cannot change after they are constructed

See docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html
Overview of Functional Programming in Java 8

• In contrast, object-oriented programming employs “hierarchical data abstraction”

See [en.wikipedia.org/wiki/Object-oriented_design](en.wikipedia.org/wiki/Object-oriented_design)
Overview of Functional Programming in Java 8

• In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
  • Components are based on stable class roles & relationships extensible via inheritance & dynamic binding

See en.wikipedia.org/wiki/Object-oriented_programming
Overview of Functional Programming in Java 8

• In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
  • Components are based on stable \textit{class} roles & relationships extensible via inheritance & dynamic binding
  • Rather than by functions that correspond to algorithmic actions

See \url{www.drdobbs.com/windows/software-complexity-bringing-order-to-ch/199901062}
Overview of Functional Programming in Java 8

- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
- Components are based on stable `class` roles & relationships extensible via inheritance & dynamic binding
- State is encapsulated by methods that perform imperative statements

```
Tree tree = ...;
Visitor printVisitor = makeVisitor(...);

for(Iterator<Tree> iter = tree.iterator();
    iter.hasNext();)
iter.next().accept(printVisitor);
```
Overview of Functional Programming in Java 8

- In contrast, object-oriented programming employs “hierarchical data abstraction”, e.g.
  - Components are based on stable class roles & relationships extensible via inheritance & dynamic binding
  - State is encapsulated by methods that perform imperative statements
  - This state is often mutable

See en.wikipedia.org/wiki/Imperative_programming

```java
Tree tree = ...;
Visitor printVisitor = 
  makeVisitor(...);

for(Iterator<Tree> iter = 
  tree.iterator();
  iter.hasNext();)
  iter.next().accept(printVisitor);
```

Access & update internal state of the iterator
Combining Object-Oriented (OO) & Functional Programming (FP) in Java 8
Benefits of Combining OO & FP in Java 8

- Java 8’s combination of functional & object-oriented paradigms is powerful!
Benefits of Combining OO & FP in Java 8

- Java 8’s functional features help close the gap between a program’s “domain intent” & its computations

See www.toptal.com/software/declarative-programming
Benefits of Combining OO & FP in Java 8

- Java 8’s functional features help close the gap between a program’s “domain intent” & its computations, e.g.,
- Domain intent defines “what”

See github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang
Benefits of Combining OO & FP in Java 8

- Java 8’s functional features help close the gap between a program’s “domain intent” & its computations, e.g.,
  - Domain intent defines “what”
  - Computations define “how”

```java
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

Download images that aren’t already cached from a list of URLs & process/store the images in parallel

Benefits of Combining OO & FP in Java 8

Java 8’s functional features help close the gap between a program’s “domain intent” & its computations, e.g.,

- Domain intent defines “what”
- Computations define “how”

```
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```
Benefits of Combining OO & FP in Java 8

• Likewise, Java 8’s object-oriented features help to structure a program’s software architecture

See [en.wikipedia.org/wiki/Software_architecture](en.wikipedia.org/wiki/Software_architecture)
Benefits of Combining OO & FP in Java 8

- Likewise, Java 8’s object-oriented features help to structure a program’s software architecture.

See sce.uhd.edu/helm/rationalunifiedprocess/process/workflow/ana_desi/co_lview.htm
Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program

Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program
- Common classes provide a reusable foundation for extensibility

See [www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf)
Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program
- Common classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies

See [www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf](www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf)
Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program
- Common classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies
- Java 8’s FP features are most effective when used to simplify computations within the context of an OO software architecture

```java
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```
Benefits of Combining OO & FP in Java 8

- e.g., consider the ImageStreamGang program
  - Common classes provide a reusable foundation for extensibility
  - Subclasses extend the common classes to create various custom implementation strategies

- Java 8’s FP features are most effective when used to simplify computations within the context of an OO software architecture
  - Especially concurrent & parallel computations

```java
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

See [docs.oracle.com/javase/tutorial/collections/streams/parallelism.html](http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html)
Benefits of Combining OO & FP in Java 8

• Since Java 8 is a hybrid language, there are situations in which mutable changes to state are allowed/encouraged

See www.infoq.com/articles/How-Functional-is-Java-8
Benefits of Combining OO & FP in Java 8

• Since Java 8 is a hybrid language, there are situations in which mutable changes to state are allowed/encouraged
  
• e.g., Java collection framework classes

See docs.oracle.com/javase/8/docs/technotes/guides/collections
Benefits of Combining OO & FP in Java 8

• However, you’re usually better off by minimizing/avoiding the use of shared mutable state in your programs!!
End of Overview of Java 8 Foundations