Learning Objectives in this Part of the Lesson

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

• Understand key aspects of functional programming
• Know how functional programming is applied in Java 8
Overview of Functional Programming
Overview of Functional Programming

- Functional programming is a “declarative” paradigm

See [en.wikipedia.org/wiki/Declarative_programming](en.wikipedia.org/wiki/Declarative_programming)
Overview of Functional Programming

- 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

```java
List<String> zap(List<String> lines, String omit) {
    return lines
        .stream()
        .filter(line ->
            !omit.equals(line))
        .collect(toList());
}
```
Overview of Functional Programming

- Conversely, object-oriented programming is an “imperative” paradigm

See en.wikipedia.org/wiki/Imperative_programming
Overview of Functional Programming

- Conversely, 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;
}
```

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

```
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;
}
```
Overview of Functional Programming

• Functional programming has its roots in lambda calculus

See en.wikipedia.org/wiki/Functional_programming
Overview of Functional Programming

- 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
Overview of Functional Programming

• 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/avoided

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

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

See [en.wikipedia.org/wiki/Object-oriented_design](http://en.wikipedia.org/wiki/Object-oriented_design)
Overview of Functional Programming

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

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

- 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

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

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

• Java 8 was released in March 2014

What's New in JDK 8

Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle's implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

• Java Programming Language
  • Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionally as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
  • Method references provide easy-to-read lambda expressions for methods that already have a name.
  • Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
  • Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type.
  • Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
  • Improved type inference.
  • Method parameter reflection.

• Collections
  • Classes in the new java.util.stream package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
  • Performance Improvement for HashMaps with Key Collisions

See www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html
Overview of Functional Programming in Java 8

• Java 8 was released in March 2014

What’s New in JDK 8

Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle’s implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

• Java Programming Language
  
  • Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
  
  • Method references provide easy-to-read lambda expressions for methods that already have a name.
  
• Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.

• Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.

• Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.

• Improved type inference.

• Method parameter reflection.

• Collections
  
  • Classes in the new java.util.stream package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.

• Performance Improvement for HashMaps with Key Collisions

[See docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]
Overview of Functional Programming in Java 8

• Java 8 was released in March 2014

What’s New in JDK 8

Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle’s implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

• Java Programming Language
  • Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
  • Method references provide easy-to-read lambda expressions for methods that already have a name.
  • Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
  • Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
  • Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
  • Improved type inference.
  • Method parameter reflection.

• Collections
  • Classes in the new java.util.stream package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
  • Performance Improvement for HashMaps with Key Collisions

See docs.oracle.com/javase/tutorial/collections/streams
Overview of Functional Programming in Java 8

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

• Java 8 is a “hybrid” that combines the functional & object-oriented paradigms

• Its functional features help close the gap between a program’s “domain intent” & its computations

See www.toptal.com/software/declarative-programming
Overview of Functional Programming in Java 8

- Java 8 is a “hybrid” that combines the functional & object-oriented paradigms
- Its functional features help close the gap between a program’s “domain intent” & its computations, e.g.,
- Domain intent defines “what”
Overview of Functional Programming in Java 8

• Java 8 is a “hybrid” that combines the functional & object-oriented paradigms

• Its 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(urlCached()))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
Overview of Functional Programming in Java 8

- Java 8 is a “hybrid” that combines the functional & object-oriented paradigms
- Its 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(urlCached()))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

Overview of Functional Programming in Java 8

- Java 8 is a “hybrid” that combines the functional & object-oriented paradigms.
- Its 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(urlCached()))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

Java 8 functional programming features connect domain intent & computations.
Overview of Functional Programming in Java 8

- Java 8 is a “hybrid” that combines the functional & object-oriented paradigms
- Its functional features help close the gap between a program’s “domain intent” & its computations
- Object-oriented features structure a program’s software architecture

See en.wikipedia.org/wiki/Software_architecture
Overview of Functional Programming in Java 8

- Java 8 is a “hybrid” that combines the functional & object-oriented paradigms.
- Its functional features help close the gap between a program’s “domain intent” & its computations.
- Object-oriented features structure a program’s software architecture, e.g.,
  - Common classes that provide a reusable foundation.

See www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf
Overview of Functional Programming in Java 8

• Java 8 is a “hybrid” that combines the functional & object-oriented paradigms
  • Its functional features help close the gap between a program’s “domain intent” & its computations
  • Object-oriented features structure a program’s software architecture, e.g.,
    • Common classes that provide a reusable foundation
    • Subclasses that extend the common classes to create various custom solutions

See www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf
Overview of Functional Programming in Java 8

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

• The Java 8 (or beyond) runtime environment (JRE) supports Java 8 features

See docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html
Overview of Functional Programming in Java 8

• The Java 8 (or beyond) runtime environment (JRE) supports Java 8 features

• Many features of Java 8 are supported by Android API level 24 (& beyond)

See android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html
Overview of Functional Programming in Java 8

• The Java 8 (or beyond) runtime environment (JRE) supports Java 8 features

• Many features of Java 8 are supported by Android API level 24 (& beyond)

• Make sure to get Android Studio 2.4 or later!

See [developer.android.com/studio/preview/features/java8-support.html](http://developer.android.com/studio/preview/features/java8-support.html)
Overview of Functional Programming in Java 8

- The LiveLessons website provides many examples of Java 8 programs

See www.github.com/douglascraigschmidt/LiveLessons
End of Overview of Basic Java 8 Features (Part 1)
Learning Objectives in this Part of the Lesson

• Understand key aspects of functional programming
• Know how functional programming is applied in Java 8
• Recognize basic functional programming features in Java 8, e.g.,
  • Lambda expressions
Learning Objectives in this Part of the Lesson

- Understand key aspects of functional programming
- Know how functional programming is applied in Java 8
- Recognize basic functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
Overview of Lambda Expressions & Method References
Overview of Lambda Expressions

- A lambda expression is an unnamed block of code (with optional parameters)

```java
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

Overview of Lambda Expressions

- A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later

```java
new Thread(() ->
    System.out.println("hello world"))
.start();
```

A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
        System.out.println("hello world"))
    .start();
```

**Lambda expressions are compact since they just focus on computation(s) to perform**
A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world"))
 .start();
```

*Lambda expressions are compact since they just focus on computation(s) to perform*
Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world"))
.start();
```

*Lambda expressions are compact since they just focus on computation(s) to perform*
Overview of Lambda Expressions

• A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

VS

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("hello world");
    }
}).start();
```

This anonymous inner class parameter requires more code to write each time
A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world");
).start();
```

VS

```java
new Thread(new Runnable() {
    public void run() {
        System.out.println("hello world");
    }
}).start();
```
Overview of Method References

- A method reference is a compact, easy-to-read handle for a method that already has a name

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reference to a static method</td>
<td>ContainingClass::staticMethodName</td>
</tr>
<tr>
<td>2. Reference to an instance method of a particular object</td>
<td>containingObject::InstanceMethodName</td>
</tr>
<tr>
<td>3. Reference to an instance method of an arbitrary object of a particular type</td>
<td>ContainingType::methodName</td>
</tr>
<tr>
<td>4. Reference to a constructor</td>
<td>ClassName::new</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)
### Overview of Method References

A method reference is a compact, easy-to-read handle for a method that already has a name.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reference to a static method</td>
<td><code>ContainingClass::staticMethodName</code></td>
</tr>
<tr>
<td>2. Reference to an instance method of a particular object</td>
<td><code>containingObject::InstanceMethodName</code></td>
</tr>
<tr>
<td>3. Reference to an instance method of an arbitrary object</td>
<td><code>ContainingType::methodName</code></td>
</tr>
<tr>
<td>of a particular type</td>
<td></td>
</tr>
<tr>
<td>4. Reference to a constructor</td>
<td><code>ClassName::new</code></td>
</tr>
</tbody>
</table>
A method reference is a compact, easy-to-read handle for a method that already has a name.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reference to a static method</td>
<td>ContainingClass::staticMethodName</td>
</tr>
<tr>
<td>2. Reference to an instance method of a particular object</td>
<td>containingObject::InstanceMethodName</td>
</tr>
<tr>
<td>3. Reference to an instance method of an arbitrary object of a particular type</td>
<td>ContainingType::methodName</td>
</tr>
<tr>
<td>4. Reference to a constructor</td>
<td>ClassName::new</td>
</tr>
</tbody>
</table>
Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>(){
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
});
```

vs

```java
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

vs

```java
Arrays.sort(nameArray, String::compareToIgnoreCase);
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex1](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex1)
Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>(){
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }});
```

```
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```

Lots of syntax for anonymous inner class
Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

```java

Arrays.sort(nameArray,
       new Comparator<String>(){
           public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
       });
```

VS

```java
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

VS

```java
Arrays.sort(nameArray, String::compareToIgnoreCase);
```

See docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html
Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
});
```

vs

```java
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

vs

```java
Arrays.sort(nameArray, String::compareToIgnoreCase);
```

Method reference is even more compact
Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

  ```java
  Arrays.sort(nameArray, new Comparator<String>() {
      public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
  });
  ```

  vs

  ```java
  Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
  ```

  vs

  ```java
  Arrays.sort(nameArray, String::compareToIgnoreCase);
  ```

It’s generally a good idea to use method references whenever you can.
The contents of a collection can be printed in various ways

```java
String[] nameArray = {
    "Barbara", "James", "Mary", "John", 
    "Robert", "Michael", "Linda", "james", "mary"};
```
• The contents of a collection can be printed in various ways

```java

• System.out.println() can be used to print out an array

```java
System.out.println(Arrays.asList(nameArray));
```

prints

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```
Overview of Method References

• The contents of a collection can be printed in various ways


• System.out.println() can be used to print out an array

• The Java 8 forEach() loop can be used to in conjunction with method references

  Stream.of(nameArray).forEach(System.out::println);

  prints

  BarbaraJamesMaryJohnLindaMichaelLindaJamesmary

End of Overview of Basic Java 8 Features (Part 2)
Overview of Basic Java 8 Features

(Part 3)

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 key aspects of functional programming
- Recognize key functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Functional interfaces
These features provide the foundation for Java 8’s concurrency mechanisms.

Learning Objectives in this Part of the Lesson

- Understand key aspects of functional programming
- Recognize key functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Functional interfaces
Overview of Functional Interfaces (Part 1)
Overview of Functional Interfaces

- A **functional interface** contains only one abstract method

```java
<<Java Interface>>
Runnable

run():void
```

There can be only one!
Overview of Functional Interfaces

• A functional interface contains only one abstract method
  
• It’s the type used for a parameter when a lambda expression or method reference is passed as an argument
Overview of Functional Interfaces

- A functional interface contains only one abstract method
- It's the type used for a parameter when a lambda expression or method reference is passed as an argument
- Java 8 defines many types of functional interfaces
A functional interface contains only one abstract method.

- It’s the type used for a parameter when a lambda expression or method reference is passed as an argument.

- Java 8 defines many types of functional interfaces.
Overview of Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - public interface Predicate<T> { boolean test(T t); }
Overview of Functional Interfaces: Predicate

- A Predicate performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

See [docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html](http://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)
A **Predicate** performs a test that returns true or false, e.g.,

```java
public interface Predicate<T> { boolean test(T t); }
```

```java
List<Integer> list =
    new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
System.out.println(list);

list.removeIf(i -> i % 2 == 0);
System.out.println(list);
```
A Predicate performs a test that returns true or false, e.g.,

```java
public interface Predicate<T> { boolean test(T t); }
```

Overview of Functional Interfaces: Predicate

List<Integer> list =
    new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
System.out.println(list);

list.removeIf(i -> i % 2 == 0);
System.out.println(list);

Pass a predicate lambda that deletes even numbers from the list
Overview of Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

```java
List<Integer> list =
    new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
System.out.println(list);
list.removeIf(i -> i % 2 == 0);
System.out.println(list);
```

- *i* is short-hand for `(Integer i)`, which leverages Java compiler’s type inference capabilities

See [docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html](http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html)
Overview of Functional Interfaces: Function

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

See [docs.oracle.com/javase/8/docs/api/java/util/function/Function.html](docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)
Overview of Functional Interfaces: Function

- A Function applies a computation on 1 parameter & returns a result, e.g.,
  - public interface Function<T, R> { R apply(T t); }
Overview of Functional Interfaces: Function

- A **Function** applies a computation on 1 parameter & returns a result, e.g.,
  
  ```java
  public interface Function<T, R> { R apply(T t); }
  
  List<Thread> threads =
      new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                      new Thread("Curly"),
                                      new Thread("Moe")));

  System.out.println(threads);
  threads.sort(Comparator.comparing(Thread::getName));
  System.out.println(threads);
  ```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex3](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex3)
Overview of Functional Interfaces: Function

- A Function applies a computation on 1 parameter & returns a result, e.g.,
  
  ```java
  public interface Function<T, R> { R apply(T t); }
  
  List<Thread> threads =
      new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                     new Thread("Curly"),
                                     new Thread("Moe")));

  System.out.println(threads);
  threads.sort(Comparator.comparing(Thread::getName));
  System.out.println(threads);
  ```
Overview of Functional Interfaces: Function

- A Function applies a computation on 1 parameter & returns a result, e.g.,
  - public interface Function<T, R> { R apply(T t); }

```java
List<Thread> threads = 
    new ArrayList<>(Arrays.asList(new Thread("Larry"),
        new Thread("Curly"),
        new Thread("Moe")));
System.out.println(threads);
threads.sort(Comparator.comparing(Thread::getName));
System.out.println(threads);
```

A method reference used to sort threads by name
Overview of Functional Interfaces: Function

• A **Function** applies a computation on 1 parameter & returns a result, e.g.,

  • public interface Function<T, R> { R apply(T t); }

  List<Thread> threads =
      new ArrayList<>(Arrays.asList(new Thread("Larry"),
          new Thread("Curly"),
          new Thread("Moe")));

  System.out.println(threads);
  threads.sort(Comparator.comparing(Thread::getName));
  System.out.println(threads);

  **A method reference used to sort threads by name**
Overview of Functional Interfaces: Function

- A `Function` applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```java
List<Thread> threads =
    new ArrayList<>((Arrays.asList(new Thread("Larry"),
        new Thread("Curly"),
        new Thread("Moe"))));
System.out.println(threads);
threads.sort(Comparator.comparing(Thread::getName));
System.out.println(threads);
```

... static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
        keyEx.apply(c1)
            .compareTo(keyEx.apply(c2));
```
A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```java
public interface Function<T, R> { R apply(T t); }
```

```
List<Thread> threads =
    new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                   new Thread("Curly"),
                                   new Thread("Moe")));
System.out.println(threads);
threads.sort(Comparator.comparing(Thread::getName));
System.out.println(threads);

... static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
        keyEx.apply(c1)
        .compareTo(keyEx.apply(c2));
    }
```

See [docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-)
Overview of Functional Interfaces: Function

- A Function applies a computation on 1 parameter & returns a result, e.g.,
  ```java
  public interface Function<T, R> { R apply(T t); }
  ```

```
List<Thread> threads =
    new ArrayList<>(Arrays.asList(new Thread("Larry"),
    new Thread("Curly"),
    new Thread("Moe")));

System.out.println(threads);
threads.sort(Comparator.comparing(Thread::getName));
System.out.println(threads);
```

```java
... static <T, U extends Comparable<? super U>> Comparator<T>
comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
        keyEx.apply(c1)
        .compareTo(keyEx.apply(c2)); }
```

See docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-
Overview of Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
  
  ```java
  public interface BiFunction<T, U, R> { R apply(T t, U u); }
  ```

See [docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html](http://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)
Overview of Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
  - public interface BiFunction<T, U, R> { R apply(T t, U u); }
A **BiFunction** applies a computation on 2 parameters & returns a result, e.g.,

```java
class BiFunction<T, U, R> { R apply(T t, U u); }
```

```java
Map<String, Integer> iqMap = new HashMap<String, Integer>() {
    { put("Larry", 100); put("Curly", 100); put("Moe", 100); }
};

iqMap.replaceAll((k, v) -> v - 50);
```

vs.

```java
for (Map.Entry<String, Integer> entry : iqMap.entrySet())
    entry.setValue(entry.getValue() - 50);
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex4](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex4)
Overview of Functional Interfaces: BiFunction

- A BiFunction applies a computation on 2 parameters & returns a result, e.g.,
- public interface BiFunction<T, U, R> { R apply(T t, U u); }

Map<String, Integer> iqMap = new HashMap<String, Integer>() {
    { put("Larry", 100); put("Curly", 100); put("Moe", 100); }
};

iqMap.replaceAll((k, v) -> v - 50);

vs.

for (Map.Entry<String, Integer> entry : iqMap.entrySet())
    entry.setValue(entry.getValue() - 50);

BiFunctional lambda subtracts 50 IQ points from each person in map
Overview of Functional Interfaces: BiFunction

- A BiFunction applies a computation on 2 parameters & returns a result, e.g.,
  
  ```java
  public interface BiFunction<T, U, R> { R apply(T t, U u); }
  ```

  ```java
  Map<String, Integer> iqMap = new HashMap<String, Integer>() {
    { put("Larry", 100); put("Curly", 100); put("Moe", 100); }
  };

  iqMap.replaceAll((k, v) -> v - 50);

  vs.

  for (Map.Entry<String, Integer> entry : iqMap.entrySet())
    entry.setValue(entry.getValue() - 50);
  ```

Conventional way of subtracting 50 IQ points from each person in map.
Overview of Functional Interfaces (Part 2)
Overview of Functional Interfaces: Consumer

- A **Consumer** accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

See [docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html](docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)
Overview of Functional Interfaces: Consumer

- A Consumer accepts a parameter & returns no results, e.g.,
  
  public interface Consumer<T> { void accept(T t); }

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html
Overview of Functional Interfaces: Consumer

- A **Consumer** accepts a parameter & returns no results, e.g.,
  
  ```java
  public interface Consumer<T> { void accept(T t); }
  ```
  
  ```java
  List<Thread> threads =
      new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                       new Thread("Curly"),
                                       new Thread("Moe")));

  threads.forEach(System.out::println);
  threads.sort(Comparator.comparing(Thread::getName));
  threads.forEach(System.out::println);
  ```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex5](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex5)
List<Thread> threads =
    new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                 new Thread("Curly"),
                                 new Thread("Moe")));

threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);

• A Consumer accepts a parameter & returns no results, e.g.,
  public interface Consumer<T> { void accept(T t); }

Overview of Functional Interfaces: Consumer
Overview of Functional Interfaces: Consumer

- A Consumer accepts a parameter & returns no results, e.g.,
  
  ```java
  public interface Consumer<T> { void accept(T t); }
  
  List<Thread> threads =
      new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                       new Thread("Curly"),
                                       new Thread("Moe")));
  
  threads.forEach(System.out::println);
  threads.sort(Comparator.comparing(Thread::getName));
  threads.forEach(System.out::println);
  ```

Overview of Functional Interfaces: Consumer

- A Consumer accepts a parameter & returns no results, e.g.,
  - public interface Consumer<T> { void accept(T t); }

```java
List<Thread> threads =
    new ArrayList<>(Arrays.asList(new Thread("Larry"),
                     new Thread("Curly"),
                     new Thread("Moe")));
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

Print out threads using forEach()
Overview of Functional Interfaces: Supplier

- A `Supplier` returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

See docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html
Overview of Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - public interface Supplier<T> { T get(); }
Overview of Functional Interfaces: Supplier

- A `Supplier` returns a value & takes no parameters, e.g.,
  
  ```java
  public interface Supplier<T> { T get(); }
  ```

  ```java
  Map<String, String> personMap = new HashMap<String, String>()
  { { put("Demon", "Naughty"); put("Angel", "Nice"); } };
  ```

  ```java
  String person = ...;
  ```

  ```java
  Optional<String> disposition =
  Optional.ofNullable(personMap.get(person));
  ```

  ```java
  System.out.println("disposition of " + person + " = " +
  disposition.orElseGet(() -> "unknown");
  ```

  Returns default value if person not found
A Supplier returns a value & takes no parameters, e.g.,

```java
public interface Supplier<T> { T get(); }
```

Map<String, String> personMap = new HashMap<String, String>() {
    { put("Demon", "Naughty"); put("Angel", "Nice"); }
};

String person = ...;

Optional<String> disposition = Optional.ofNullable(personMap.get(person));

System.out.println("disposition of " + person + " = " + disposition.orElseGet(() -> "unknown"));

Returns default value if person not found
A **Supplier** returns a value & takes no parameters, e.g.,

```
public interface Supplier<T> { T get(); }
```

Map<String, String> personMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };

String person = ...;

Optional<String> disposition =
    Optional.ofNullable(personMap.get(person));

System.out.println("disposition of "
    + person + " = "
    + disposition.orElseGet(() -> "unknown"));

See [docs.oracle.com/javase/8/docs/api/java/util/Optional.html](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)
Overview of Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```java
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex7](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex7)
Overview of Functional Interfaces: Supplier

- A **Supplier** returns a value & takes no parameters, e.g.,
- public interface Supplier<T> { T get(); }

```java
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

Overview of Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - public interface Supplier<T> { T get(); }

```java
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

A `Supplier` returns a value & takes no parameters, e.g.,

```java
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }
    private String hello() { return "hello";
    }
}
```

Overview of Functional Interfaces: Supplier

- A **Supplier** returns a value & takes no parameters, e.g.,
  - public interface Supplier<T> { T get(); }

```java
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

*get() creates a new CrDemo object using a “constructor reference”*
A Supplier returns a value & takes no parameters, e.g.,

```java
public interface Supplier<T> { T get(); }

class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```
A functional interface may also have default methods or static methods

```java
interface Comparator {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed()
    { return Collections.reverseOrder(this); } }

static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
{ return Collections.reverseOrder(); } ...
```

See docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
A functional interface may also have default methods or static methods, e.g.

```java
interface Comparator {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }

    static <T extends Comparable<? super T>> Comparator<T> reverseOrder() {
        return Collections.reverseOrder();
    }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/Comparator.html](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)
A functional interface may also have default methods or static methods, e.g.,

```java
interface Comparator {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }

    static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder() {
        return Collections.reverseOrder();
    }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/Comparator.html](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)
A functional interface may also have default methods or static methods, e.g.,

```java
interface Comparator {
    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed()
    { return Collections.reverseOrder(this); }

    static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder()
    { return Collections.reverseOrder(); }
}
...
A functional interface may also have default methods or static methods, e.g.,

```java
interface Comparator {
    int compare(T o1, T o2);
}

boolean equals(Object obj);

default Comparator<T> reversed()
    { return Collections.reverseOrder(this); }

static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder()
    { return Collections.reverseOrder(); }
...
```

See [docs.oracle.com/javase/8/docs/api/java/util/Comparator.html](docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)
Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```java
interface Comparator {
    int compare(T o1, T o2);
    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }

    static <T extends Comparable<? super T>> Comparator<T> reverseOrder() {
        return Collections.reverseOrder();
    }
    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
    int compare(T o1, T o2);
    boolean equals(Object obj);

    default Comparator<T> reversed()
    { return Collections.reverseOrder(this); }

    static <T extends Comparable<? super T>>
    Comparator<T> reverseOrder()
    { return Collections.reverseOrder(); }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html](docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

An abstract method that overrides a public `java.lang.Object` method does not count as part of the interface's abstract method count.
End of Overview of Basic Java 8 Features (Part 3)