

## Session 1: Review Answers

1. An interface may have at most one default method. [T/F]
  - **False**: An interface can have an unlimited number of default methods
2. A default method in an interface can call other methods in the same interface. [T/F]
  - **True**: We did this in the lab, where `silence()` calls `setVolume()` <sup>(1)</sup>
3. Explain how default methods help interfaces to evolve "cleanly," i.e., without breaking existing code.
  - When adding a new default method, **existing implementations continue working unchanged**
  - If you add a normal (abstract) method, all implementations must add it
4. Functional interfaces must include the `@FunctionalInterface` annotation. [T/F]
  - **False**: It is optional, but does add safety as the compiler will check
5. The `Comparator` interface has **2** abstract methods, yet it's a functional interface. Explain.
  - One of the methods is `equals()` - **defined in class `Object`**, so doesn't count

### Notes:

- <sup>(1)</sup> All the methods defined in an interface are available to a default method
- In particular, regular (abstract) interface methods are available - when they are called, the method in the implementing class for that particular instance will be called
  - This is just normal polymorphism in action

## Session 1: Review Answers

6. A static interface method can only return an instance of that interface type, e.g., `Shape.get()` must have `Shape` as its return type. [T/F]
- **False**: static interface methods can return instances of any type. However, they are often used as factory methods that do return an instance by the interface type
7. What's wrong with the interface below?
- The method `payTaxes()` has a body, but is a regular (abstract) interface method, so this is illegal and won't compile
  - If you want a default method, it must include the default keyword, e.g.

**default** public double payTaxes() { /\* ... \*/ }

```
public interface TaxPayer {  
    public void register();  
    public double payTaxes() {  
        return 0.0;  
    }  
}
```

### Notes:

## Session 2: Review Answers

1. What is the motivation for lambdas?
  - To create more compact code
  - By encapsulating a single unit of behavior, and passing it to other code
2. Explain the relationship between functional interfaces and lambdas.
  - Lambdas implement the single abstract method in a functional interface
  - Can be passed to anything expecting the appropriate functional interface type
3. Lambdas can throw exceptions. [T/F]
  - **True**: Lambdas can throw exceptions compatible with their functional interface method definition.
4. Lambdas exist in an enclosing class, but don't have access to the variables and methods in that class. [T/F]
  - **False**: Lambdas can access variables of the enclosing class (variable capture) as well as instance and static methods.
5. Given this abstract method: **Date toDate(String dateString);**  
Sketch out a compatible lambda. (Not a full implementation, just a sketch.)  
**dateString -> Date.valueOf(dateString)**

### Notes:

## Session 2: Review Answers

6. Given this lambda: **input -> Math.sqrt(input)**

Write an equivalent method reference.

**Math::sqrt**

7. Given this lambda:

**(String name, String prefix) -> name.contains(prefix)**

Write an equivalent method reference.

**String::contains**

8. Given this lambda: **dept -> dept.getLocation()**

Write an equivalent method reference (dept is a Department instance)

**Department::getLocation**

9. Why can you usually omit the parameter types in a lambda?

–They can generally be inferred from the functional interface "behind" it

10. Is the following lambda compatible with this method: **void foo(int x)?**

**x -> x \* 2**

–**No**: the shown lambda returns a value, and the method is of type void

### Notes:

## Session 3: Review Answers

1. Stream methods `peek()` and `forEach()` both take a Consumer function; what's the difference between them?
  - `peek()` is an **intermediate operation**, allowing for chaining
  - `forEach()` is a **terminal operation** (which returns no result)
2. Describe the "anatomy" of a stream.
  - Input source (e.g. collection) → Stream → Intermediate ops → terminal op
3. Function descriptor and general purpose of Functional Interfaces:
  - Predicate: `T -> boolean`      takes `T`, returns **boolean**
    - Define a true/false condition on an instance of `T`
  - Function: `T -> R`      takes `T`, returns `R`
    - Map input elements `T` to output elements `R`
  - Comparator: `(T,T) -> int`      takes `T, T`, returns **int**
    - **Compare** two instance of `T`, and impose an **ordering**
  - Consumer: `T -> void`      takes `T`, returns nothing
    - Do something with input argument
  - Supplier: `() -> T`      takes nothing, returns `T`
    - Provide an object or value

Continued on next page

### Notes:

## Session 3: Review Answers

4. `Stream.map(Function<T,R>)` returns a `Map<T,R>`. [T/F]
  - **False**: It returns a `Stream<R>` since the function maps from T to R
5. Describe partitioning and grouping collectors.
  - **Partitioning** collectors partition into a **2-row** true/false map
  - **Grouping** collectors groups into an **N-row** map keyed by an attribute
6. What is comparator chaining?
  - Allows you to specify multiple comparators, applied one after the other
  - e.g. - sort by salary, and then sort by name within a given salary
7. What do we mean by the term "downstream collector?"
  - A downstream collector is used in conjunction with another collector - usually a partitioning/grouping collector. They post-process the partition or group.
8. A `Collector<T,A,R>` produces an object of type \_\_\_\_\_.
  - **R**!

Continued on next page

Notes:

## Session 3: Review Answers

9. What is the difference between `Collector` and `Collectors`?
  - `Collectors` is a class that has **factory methods** for `Collector` instances
10. Difference between an intermediate operation and a terminal operation?
  - An intermediate operation results in a stream that you can continue processing on (e.g. `filter()`)
  - A terminal operation ends the pipeline. It either produces a result (e.g. `collect()`), or processes the items (e.g. `forEach()`)

### Notes:

## Session 4: Review Answers

1. Streams are the only mechanism in Java to do parallel processing. [T/F]
  - **False**: You can, for example, use the Fork/join framework directly
2. How do you enable parallel processing of a stream?
  - Call `parallel()` on the stream to indicate you want parallel processing
3. Executing a stream pipeline in parallel is generally preferred over serial execution, because it's faster. [T/F]
  - **False**: Only pipeline in parallel if it's safe and results in better performance
4. List the "requirements" for parallel stream processing.
  - Operations on each element must be **independent** of those on other elements
  - **Non-interference**: No changing of underlying element source
  - **No sharing** of mutable state
5. What is the difference between stateless and stateful element operations when executing a stream pipeline?
  - **Stateless** operations retain no state from previous elements (e.g. `map()`)
  - **Stateful** operations do depend on previous elements (e.g. `distinct()`)
  - Stateless operations are preferred for parallel processing

### Notes:



## Session 4: Review Answers

6. What is the principle of non-interference? Does it apply to sequential streams, parallel streams, or both?
  - No changing of the underlying element source while using its stream
  - This is true for both sequential and parallel streams
7. Describe a race condition. You may use a human situation / analogy.
  - One person makes a deposit into a joint bank account, while another withdraws
  - They are both accessing the same mutable data in parallel
8. Even if your use case meets the parallel processing "requirements," you may not benefit from parallelism. Why not?
  - Many reasons - e.g. you may not require enough processing to make up for the overhead involved in parallel processing. Or you have no multicore machine !
9. Concurrent modification of a shared resource is not possible with parallel streams, because you will get inconsistent (incorrect) results. [T/F]
  - **False**: You can synchronize the access - though this will reduce the benefits
10. The only way to truly know if a given task will benefit from parallelism is to run different test cases and compare the results. [T/F]
  - **True**: Analysis can give you a guideline, but it's subtle and complex, so test it

### Notes:

## Session 5: Review Answers

1. `ZonedDateTime` is the only time-zone-aware date/time type. [T/F]
  - **True**
2. How would you create one of the Java 8 date/time types from a `java.util.Calendar` or `java.util.Date` object?
  - There are several choices - we list several here
  - Both have a **`toInstant()`** method that returns an `Instant` directly
  - `Calendar` lets you get the values for each field (e.g. day, year, etc) from which you can create Java 8 date/time instances
  - You can use **`Date.toString()`** to get its string representation, then parse it using a Java 8 type's **`parse()`** and an appropriate formatter
3. What's the difference between the interval types `Period` and `Duration`?
  - A **`Period`** is a **date-based** interval (years, months, days)
  - A **`Duration`** is a **time-based** interval (seconds + nanoseconds)
4. Describe some of the API patterns in the `java.time` package.
  - **`now()`**: Gets the value of now
  - **`of()`**: Gets a value based on individual elements passed in
  - **`parse()`**: Get a value by parsing an input string

### Notes:

## Session 5: Review Answers

5. Explain the difference between parsing and formatting dates/times.

Which class is used in each activity?

- **Parsing** takes an input string and converts to a date/time
- **Formatting** specifies the format of a string representation of a date/time
- **DateTimeFormatter** can be used with both parsing (to specify the formatting of the incoming string) and formatting output (to specify the formatting of the generated string)

6. The following two time-zones are equivalent: [T/F]

- **False**: Region-based ("America/Chicago") accommodate daylight savings time
- Offset-based do not take DST into account

```
ZoneId chicago = ZoneId.of("America/Chicago");  
ZoneId central = ZoneId.of("UTC-06:00");
```

7. All the date/time types are immutable and thread-safe. [T/F]

- **True**

8. List some common use cases for Period and Duration.

- Get amount of time between two dates/times
- Add/subtract an amount of time to a date/time

### Notes: