Higher-Order Functions Exercises in Scala

Basic Higher-Order Function Exercises

1. **Create a Function that Accepts Another Function**:

```
def applyTwice(f: Int => Int, x: Int): Int
```

2. **Implement a Function to Transform a List**:

```
def transform(lst: List[Int], f: Int => Int): List[
    Int]
```

3. **Combine Two Functions**:

```
def compose[A, B, C](f: A => B, g: B => C): A => C
```

Intermediate Exercises

4. **Custom Map Function**:

```
def customMap[A, B](lst: List[A], f: A => B): List[B
    ]
```

5. **Filter Using HOF**:

```
def customFilter[A](lst: List[A], predicate: A =>
    Boolean): List[A]
```

6. **Reduce/ Fold Implementation**:

```
def customReduce[A](lst: List[A], f: (A, A) => A): A
```

7. **Partition Elements**:

```
def partition[A](lst: List[A], predicate: A =>
    Boolean): (List[A], List[A])
```

Advanced Exercises

8. **Implement a Generic Sort**:

```
def sort[A](lst: List[A])(compare: (A, A) => Boolean
    ): List[A]
```

9. **Function Composition for Lists**: Chain multiple transformations on a list using `map` and `flatMap`. For example:
   - Add 2 to each element.
   - Filter only even numbers.
   - Multiply remaining numbers by 3.

10. **Curried Functions**:

```
def addCurried(x: Int)(y: Int): Int
```

11. **Create a Pipeline**:

```
val f1: Int => Int = _ + 1
val f2: Int => Int = _ * 2
val pipeline = f1 andThen f2
```

12. **FlatMap on Nested Structures**:

```
val nested = List(List(1, 2), List(3, 4))
val flattened = nested.flatMap(identity)
```

Real-World Use Case Exercises

13. **Custom JSON Serializer**: Write a function to serialize a case class to JSON using `map` and `reduce`.

14. **Data Aggregation**: Given a list of transactions, filter out those above a threshold and compute their total:

```
case class Transaction(id: Int, amount: Double)
def totalAboveThreshold(transactions: List[
    Transaction], threshold: Double): Double
```

15. **Group By Using Fold**:

```
def groupBy[A, K](lst: List[A])(keyFunc: A => K):
    Map[K, List[A]]
```

16. **Function as a Parameter**:

```
def timeExecution[A](block: => A): A
```

Challenging Exercises

17. **Build a Functional Pipeline**: Create a pipeline of transformations that filters data, transforms it, and aggregates results using a fold.

18. **Custom Higher-Order Function**:

```
def retry[T](times: Int)(block: => T): T
```

19. **Higher-Order Functions in Recursion**:

```
def recursiveSum[A](lst: List[A], f: A => Int): Int
```