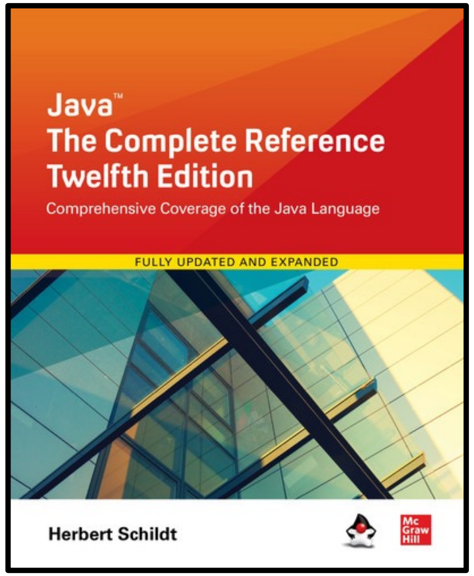
The aggregate function

from sequential and parallel folds to parallel aggregation

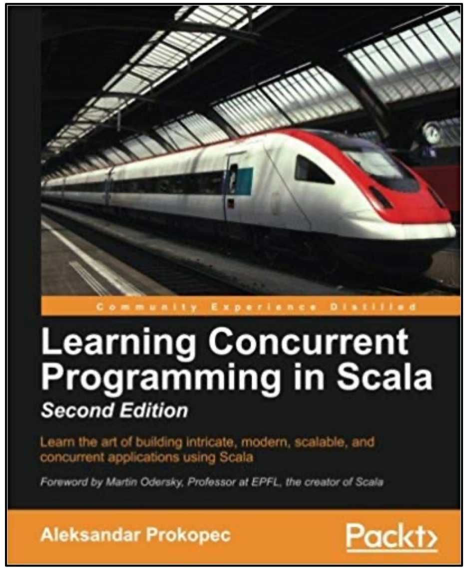
Java and Scala

based on



Herb Schildt
https://www.herbschildt.com/

Aleksandar Prokopec
@alexprokopec



slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Reduction Operations

Consider the **min( )** and **max( )** methods in the preceding example program. Both are **terminal operations** that return a result based on the elements in the **stream**.

In the language of the **stream API**, they represent ***reduction operations*** because **each reduces a stream to a single value**—in this case, the minimum and maximum.

The **stream API** refers to these as *special case* **reductions** because they perform a specific function.

In addition to **min( )** and **max( )**, other special case **reductions** are also available, such as **count( )**, which counts the number of elements in a **stream**.

However, **the stream API generalizes this concept by providing the reduce( ) method. By using reduce( ), you can return a value from a stream based on any arbitrary criteria.**

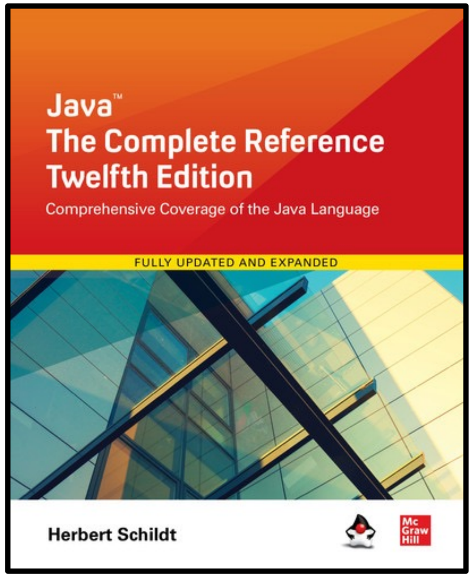
By definition, all reduction operations are terminal operations.

Stream defines three versions of reduce( ). The two we will use first are shown here:

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identityVal, BinaryOperator<T> accumulator)
```

The first form returns an object of type **Optional**, which contains the result. The second form returns an object of type **T** (which is the element type of the stream).

In both forms, *accumulator* is a function that operates on two values and produces a result. In the second form, *identityVal* is a value such that an accumulator operation involving *identityVal* and any element of the stream yields that element, unchanged.



Herb Schildt

For example, if the operation is addition, then the identity value will be 0 because 0 + x is x.

For multiplication, the identity value will be 1, because 1 * x is x.

BinaryOperator is a functional interface declared in java.util.function that extends the BiFunction functional interface.

BiFunction defines this abstract method:

```
R apply(T val, U val2)
```

Here, R specifies the result type, T is the type of the first operand, and U is the type of second operand.

Thus, apply( ) applies a function to its two operands (*val* and *val2*) and returns the result.

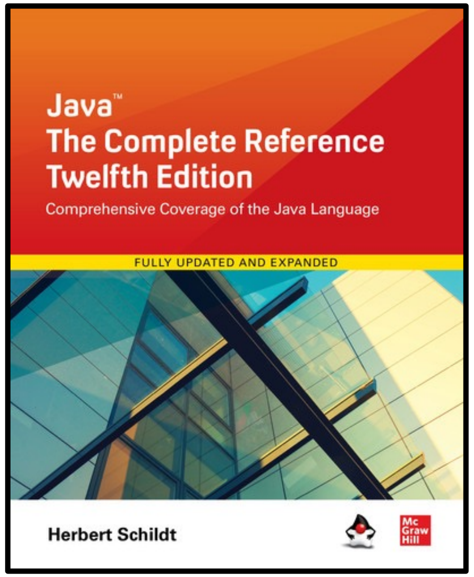
When BinaryOperator extends BiFunction, it specifies the same type for all the type parameters.

Thus, as it relates to BinaryOperator, apply( ) looks like this:

```
T apply(T val, T val2)
```

Furthermore, as it relates to reduce( ), *val* will contain the previous result and *val2* will contain the next element.

In its first invocation, val will contain either the identity value or the first element, depending on which version of reduce( ) is used.



Herb Schildt

It is important to understand that the accumulator operation must satisfy three constraints.

It must be

- Stateless
- Non-interfering
- Associative

As explained earlier, ***stateless*** **means that the operation does not rely on any state information. Thus, each element is processed independently.**

Non-interfering **means that the data source is not modified by the operation**.

Finally, the operation must be *associative*.

Here, the term *associative* is used in its normal, arithmetic sense, which means that, given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first.

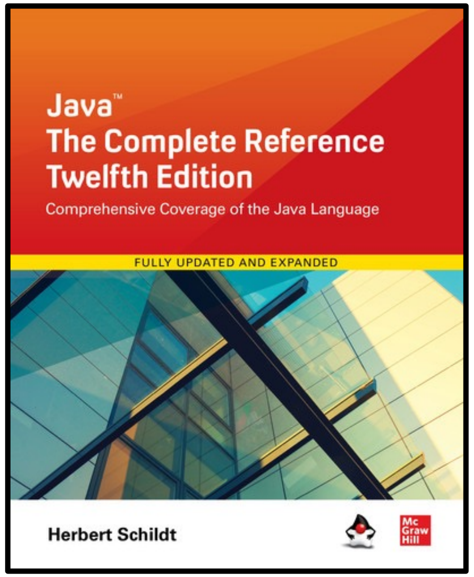
For example,

```
(10 * 2) * 7
```

yields the same result as

```
10 * (2 * 7)
```

Associativity is of particular importance to the use of reduction operations on parallel streams, discussed in the next section.



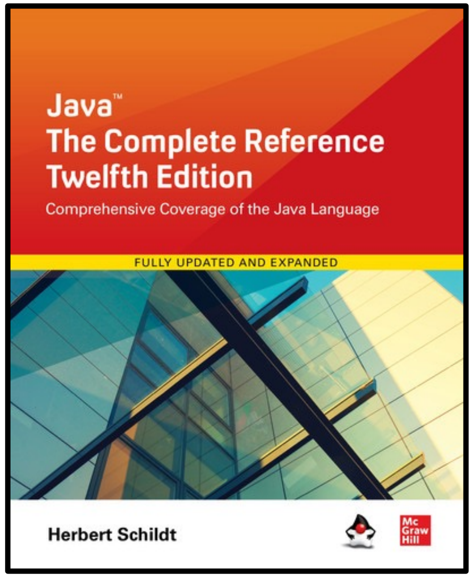
Herb Schildt

The following program demonstrates the versions of reduce( ) just described:

```
import java.util.*;
import java.util.stream.*;

public class StreamDemo2 {

  public static void main(String[] args) {

    // Create a list of integer values
    ArrayList<Integer> myList = new ArrayList<>();

    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    // Two ways to obtain the integer product of the elements in myList by use of reduce().
    Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
    if (productObj.isPresent())
      System.out.println("Product as Optional: " + productObj.get());

    int product = myList.stream().reduce(1, (a,b) -> a*b);
    System.out.println("Product as int: " + product);
  }
}
```

As the output here shows, both uses of reduce( ) produce the same result:

```
  Product as Optional: 2570400
  Product as int: 2570400
```



Herb Schildt

In the program, the first version of reduce( ) uses the lambda expression to produce a product of two values.

```
Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
```

In this case, because the stream contains Integer values, the Integer objects are automatically unboxed for the multiplication and reboxed to return the result.

The two values represent the current value of the running result and the next element in the stream. The final result is returned in an object of type Optional.

The value is obtained by calling get( ) on the returned object.

In the second version, the identity value is explicitly specified, which for multiplication is 1. Notice that the result is returned as an object of the element type, which is Integer in this case.

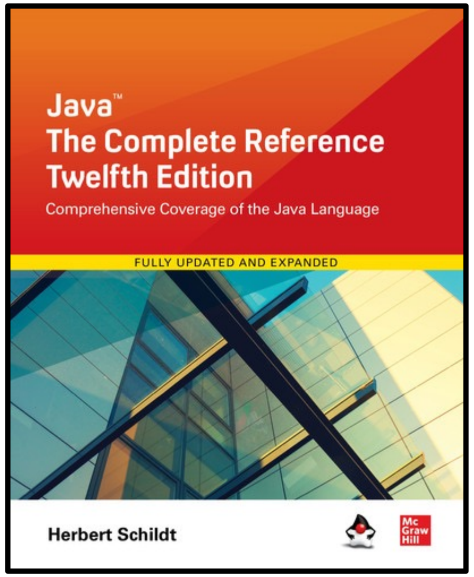
```
int product = myList.stream().reduce(1, (a,b) -> a*b);
```

Although simple reduction operations such as multiplication are useful for examples, reductions are not limited in this regard.

For example, assuming the preceding program, the following obtains the product of only the even values:

```
int evenProduct = myList.stream().reduce(1, (a,b) -> {
                      if (b%2 == 0) return a*b; else return a;
                    });
```

Pay special attention to the lambda expression. If b is even, then a * b is returned. Otherwise, a is returned. This works because a holds the current result and b holds the next element, as explained earlier.



Herb Schildt

In **Scala**, this version of the **reduce** function is called **reduceOption**.

Java

```
Optional<T> reduce(BinaryOperator<T> accumulator)

interface BinaryOperator<T> extends BiFunction<T, T, T>

interface BiFunction<T, U, R>
```

Scala

```
def reduceOption[B >: A](op: (B, B) => B): Option[B]
```

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream){
  if (!foundAny) {
    foundAny = true;
    result = element;
  } else result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

but is not constrained to execute sequentially. The accumulator function must be an associative function.

This is a terminal operation.

Parameters:
accumulator - an associative, non-interfering, stateless function for combining two values

Returns:
an `Optional` describing the result of the reduction

Throws:
`NullPointerException` - if the result of the reduction is null

See Also:
`reduce(Object, BinaryOperator)`
`min(Comparator)`
`max(Comparator)`

Java

Module java.base
Package java.util.stream

```
interface Stream<T>
```

Type Parameters: T - the type of the stream elements

```
def reduceOption[B >: A](op: (B, B) => B): Option[B]
```

Reduces the elements of this collection, if any, using the specified associative binary operator.

The order in which operations are performed on elements is unspecified and may be nondeterministic.

Type parameters: B A type parameter for the binary operator, a supertype of A.

Value parameters: op A binary operator that must be associative.

Returns: An option value containing result of applying reduce operator op between all the elements if the collection is nonempty, and None otherwise.

Source: `IterableOnce.scala`

Scala

Scala 3/scala.collection/IterableOnceOps

```
trait IterableOnceOps[+A, +CC[_], +C]
```

This implementation trait can be mixed into an IterableOnce to get the basic methods that are shared between Iterator and Iterable.

```scala
def reduceOption[B >: A](op: (B, B) => B): Option[B] =
  reduceLeftOption(op)
```

Scala

```scala
def reduceLeftOption[B >: A](op: (B, A) => B): Option[B] =
  if (isEmpty) None else Some(reduceLeft(op))
```

```scala
def reduceLeft[B >: A](op: (B, A) => B): B =
```

Applies a **binary operator** to all elements of this collection, going **left** to **right**.

Note: will **not terminate** for **infinite-sized** collections. **Note**: **might** return **different results** for **different runs**, unless the underlying collection type is **ordered** or the **operator** is **associative** and **commutative**.

Params:
op the **binary operator**.

Type parameters:
B the result type of the **binary operator**.

Returns:
the result of **inserting op between consecutive elements** of this collection, going **left** to **right**:

$$op(op(\ldots op(x_1, x_2,,) \ldots, x_{n-1}), x_n)$$

where $x_1, \ldots, x_n$ are the elements of this collection.

Throws: UnsupportedOperationException – if this collection is **empty**.
Source: `IterableOnce.scala`

```scala
def reduce[B >: A](op: (B, B) => B): B = reduceLeft(op)
```

Reduces the elements of this collection using the specified **associative binary operator**.

The **order** in which operations are performed on elements is **unspecified** and may be **nondeterministic**.

Params:

B A type parameter for the **binary operator**, a supertype of A.
op A **binary operator** that must be **associative**.

Returns:
The result of applying **reduce operator op** between all the elements if the collection is nonempty.

Throws: UnsupportedOperationException – if this collection is **empty**.
Source: `IterableOnce.scala`

In **Scala**, this version of the **reduce** function is called **fold**.

Java

```java
T reduce(T identityVal, BinaryOperator<T> accumulator)

interface BinaryOperator<T> extends BiFunction<T, T, T>

interface BiFunction<T, U, R>
```

Scala

```scala
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

```java
T reduce(T identity, BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

```java
    T result = identity;
    for (T element : this stream)
        result = accumulator.apply(result, element)
    return result;
```

but is not constrained to execute sequentially.

The identity value must be an identity for the accumulator function. This means that for all t, accumulator.apply(identity, t) is equal to t. The accumulator function must be an associative function. This is a terminal operation.

Params:
identity – the identity value for the accumulating function
accumulator – an associative, non-interfering, stateless function for combining two values

Returns:
the result of the reduction

API Note:
Sum, min, max, average, and string concatenation are all special cases of reduction. Summing a stream of numbers can be expressed as:

```java
    Integer sum = integers.reduce(0, (a, b) -> a+b);
```

or:

```java
    Integer sum = integers.reduce(0, Integer::sum);
```

While this may seem a more roundabout way to perform an aggregation compared to simply mutating a running total in a loop, reduction operations parallelize more gracefully, without needing additional synchronization and with greatly reduced risk of data races.

Module java.base
Package java.util.stream

```java
interface Stream<T>
```

Type Parameters: T - the type of the stream elements

```scala
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

Folds the elements of this collection using the specified associative binary operator. The default implementation in IterableOnce is equivalent to foldLeft but may be overridden for more efficient traversal orders.

The order in which operations are performed on elements is unspecified and may be nondeterministic. Note: will not terminate for infinite-sized collections.

Type parameters: A1 a type parameter for the binary operator, a supertype of A.

Value parameters: op a binary operator that must be associative.

z a neutral element for the fold operation; may be added to the result an arbitrary number of times, and must not change the result (e.g., Nil for list concatenation, 0 for addition, or 1 for multiplication).

Returns: the result of applying the fold operator op between all the elements and z, or z if this collection is empty.

Source: `IterableOnce.scala`

Scala 3/scala.collection/IterableOnceOps

```scala
trait IterableOnceOps[+A, +CC[_], +C]
```

This implementation trait can be mixed into an IterableOnce to get the basic methods that are shared between Iterator and Iterable.

```scala
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

```scala
def foldLeft[B](z: B)(op: (B, A) => B): B
```

Applies a **binary operator** to a **start value** and all elements of this collection, going **left** to **right**.

Note: will **not terminate** for **infinite-sized** collections.

Note: **might** return **different results** for **different runs**, unless the underlying collection type is **ordered** or the **operator** is **associative** and **commutative**.

Params:
z – the **start value**.
op – the **binary operator**.

Type parameters:
B – the result type of the **binary operator**.

Returns:
the result of **inserting op between consecutive elements** of this collection, going **left** to **right** with the **start value z** on the **left**:

```
op(...op(z, x1,), x2, ..., xn,)
```

where $x_1, \ldots, x_n$, are the elements of this collection. Returns **z** if this collection is empty.

Scala

```scala
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

By the way, speaking of the above **fold** function, the **z** (**neutral element** – **unit** - **zero**) and **associative binary operator op** form a **monoid**, so libraries like **Cats** define an alternative **fold** function (with alias **combineAll**, to avoid clashes with the above **fold** function) that operates on **monoids**.

@philip_schwarz

```scala
def fold[A](fa: F[A])(implicit A: Monoid[A]): A =
  foldLeft(fa, A.empty) { (acc, a) => A.combine(acc, a) }
```

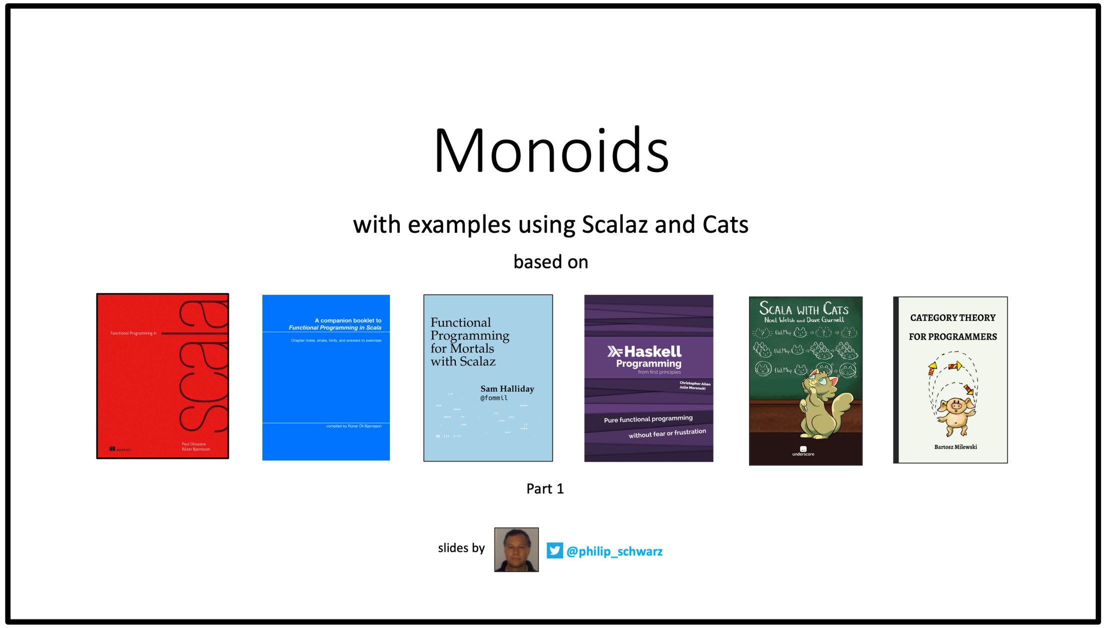
```scala
def combineAll[A: Monoid](fa: F[A]): A =
  fold(fa)
```

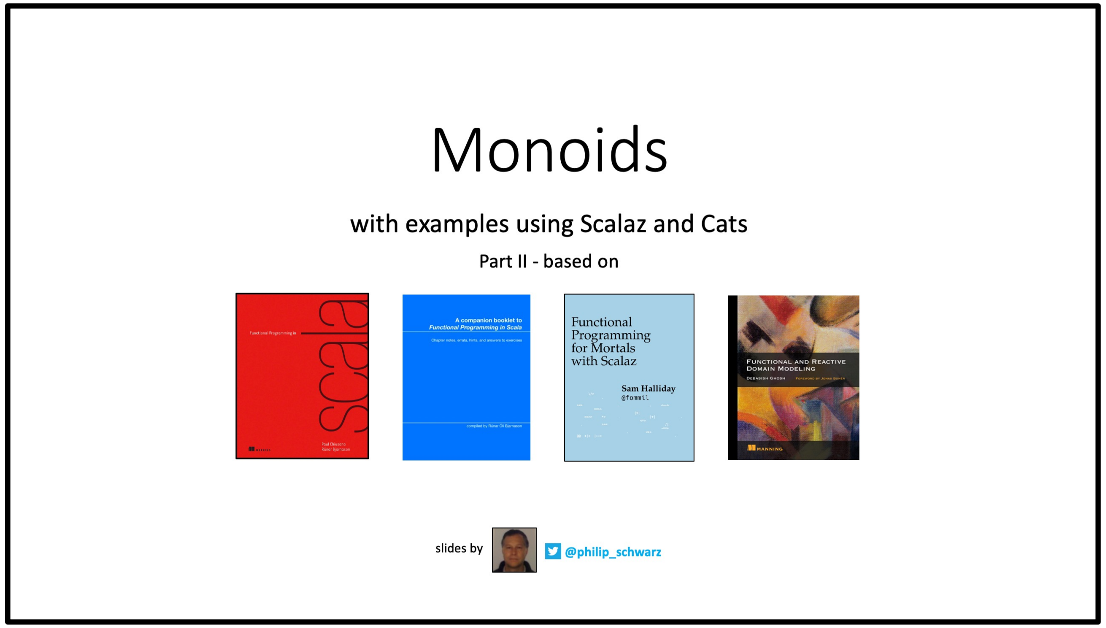
```scala
trait Monoid[A] {
  def combine(x: A, y: A): A
  def empty: A
}
```

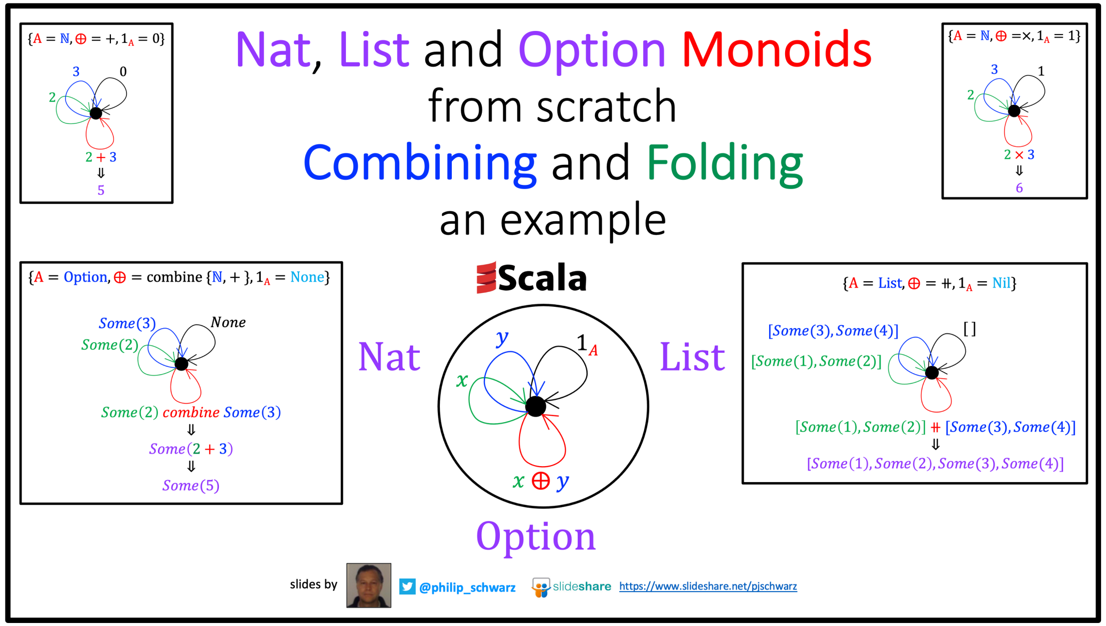
Cats

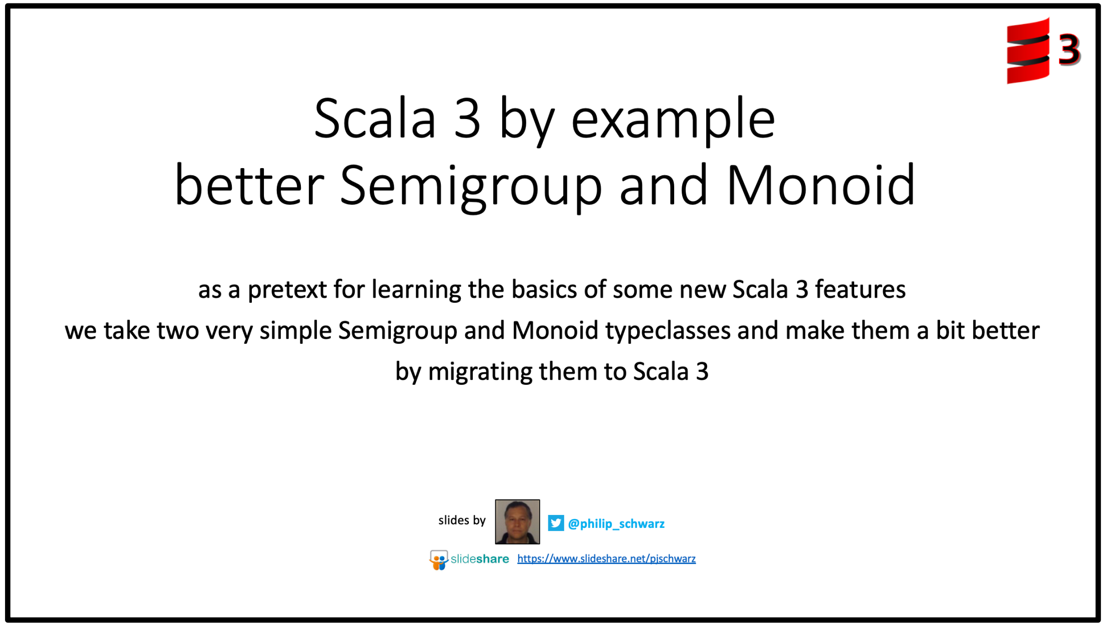
```scala
import cats.Monoid
import cats.Foldable
import cats.instances.int._
import cats.instances.string._
import cats.instances.option._
import cats.instances.list._
import cats.syntax.foldable._

assert( List(1,2,3).combineAll == 6 )
assert( List("a","b","c").combineAll == "abc" )
assert( List(List(1,2),List(3,4),List(5,6)).combineAll == List(1,2,3,4,5,6) )
assert( List(Some(2), None, Some(3), None, Some(4)).combineAll == Some(9) )
```

If you want to know more about **monoids**, here are some related decks.









Using Parallel Streams

Before exploring any more of the stream API, it will be helpful to discuss parallel streams.

As has been pointed out previously in this book, the parallel execution of code via multicore processors can result in a substantial increase in performance.

Because of this, parallel programming has become an important part of the modern programmer's job. However, parallel programming can be complex and error-prone.

One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

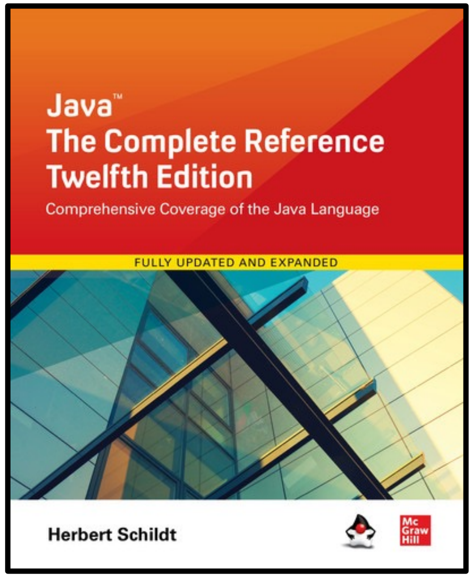
Parallel processing of a stream is quite simple to request: just use a parallel stream. As mentioned earlier, one way to obtain a parallel stream is to use the parallelStream( ) method defined by Collection.

Another way to obtain a parallel stream is to call the parallel( ) method on a sequential stream. The parallel( ) method is defined by BaseStream, as shown here:

```
S parallel()
```

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.) Understand, of course, that even with a parallel stream, parallelism will be achieved only if the environment supports it.

Once a parallel stream has been obtained, operations on the stream can occur in parallel, assuming that parallelism is supported by the environment.



Herb Schildt

Caveats with parallel collections

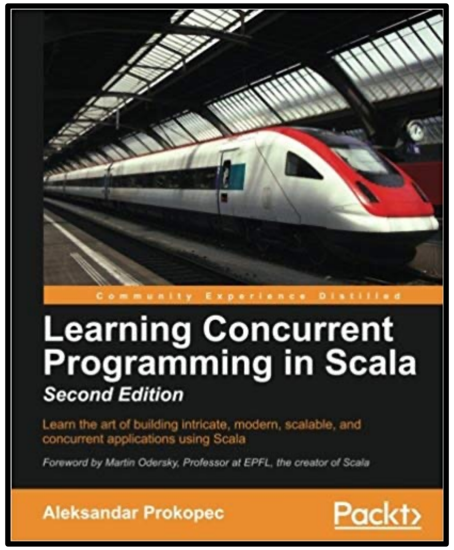
Parallel collections were designed to provide a programming API similar to sequential Scala collections. Every sequential collection has a parallel counterpart and most operations have the same signature in both sequential and parallel collections. Still, there are some caveats when using parallel collections, and we will study them in this section.

Non-parallelizable collections

Parallel collections use splitters, represented with the Splitter[T] type, in order to provide parallel operations. A splitter is a more advanced form of an iterator; in addition to the iterator's next and hasNext methods, splitters define the split method, which divides the splitter S into a sequence of splitters that traverse parts of the S splitter:

```
def split: Seq[Splitter[T]]
```

This method allows separate processors to traverse separate parts of the input collection. The split method must be implemented efficiently, as this method is invoked many times during the execution of a parallel operation. In the vocabulary of computational complexity theory, the allowed asymptotic running time of the split method is O(log (N)), where N is the number of elements in the splitter. Splitters can be implemented for flat data structures such as arrays and hash tables, and tree-like data structures such as immutable hash maps and vectors. Linear data structures such as the Scala List and Stream collections cannot efficiently implement the split method. Dividing a long linked list of nodes into two parts requires traversing these nodes, which takes a time that is proportionate to the size of the collection.



Aleksandar Prokopec
@alexprokopec

Operations on Scala collections such as Array, ArrayBuffer, mutable HashMap and HashSet, Range, Vector, immutable HashMap and HashSet, and concurrent TrieMap can be parallelized. We call these collections *parallelizable*.

Calling the par method on these collections creates a parallel collection that shares the same underlying dataset as the original collection. No elements are copied and the conversion is fast.

Other Scala collections need to be converted to their parallel counterparts upon calling par. We can refer to them as *non-parallelizable collections*. Calling the par method on non-parallelizable collections entails copying their elements into a new collection. For example, the List collection needs to be copied to a Vector collection when the par method is called, as shown in the following code snippet:

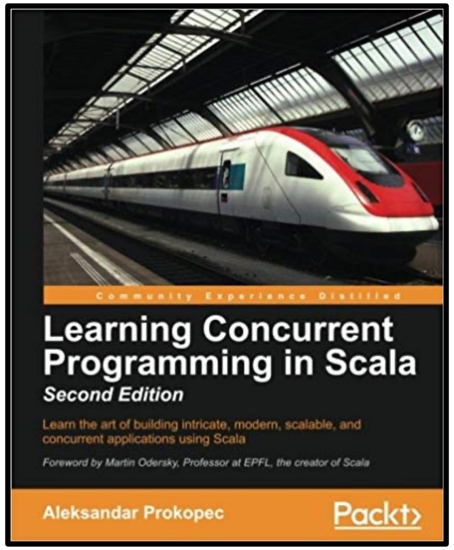
```
object ParNonParallelizableCollections extends App {
  val list = List.fill(1000000)("")
  val vector = Vector.fill(1000000)("")
  log(s"list conversion time: ${timed(list.par)} ms")
  log(s"vector conversion time: ${timed(vector.par)} ms")
}
```

Calling par on List takes 55 milliseconds on our machine, whereas calling par on Vector takes 0.025 milliseconds. Importantly, the conversion from a sequential collection to a parallel one is not itself parallelized, and is a possible sequential bottleneck.

TIP

> Converting a non-parallelizable sequential collection to a parallel collection is not a parallel operation; it executes on the caller thread.

Sometimes, the cost of converting a non-parallelizable collection to a parallel one is acceptable. If the amount of work in the parallel operation far exceeds the cost of converting the collection, then we can bite the bullet and pay the cost of the conversion. Otherwise, it is more prudent to keep the program data in parallelizable collections and benefit from fast conversions. When in doubt, measure!



Aleksandar Prokopec
@alexprokopec

For example, the first reduce( ) operation in the preceding program can be parallelized by substituting parallelStream( ) for the call to stream( ):

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b) -> a*b);
```

The results will be the same, but the multiplications can occur in different threads.

As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative.

This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

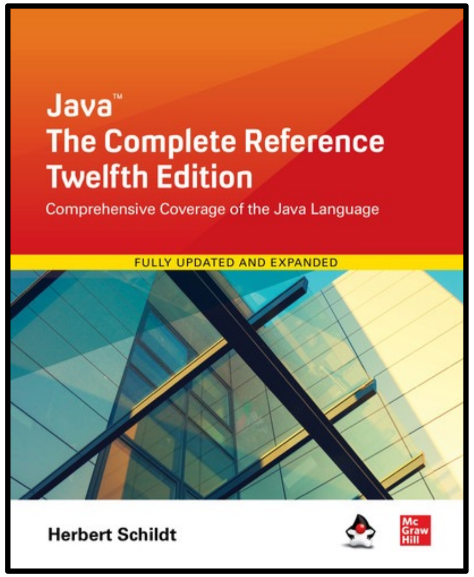
When using parallel streams, you might find the following version of reduce( ) especially helpful. It gives you a way to specify how partial results are combined:

```
<U> U reduce(U identityVal,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

In this version, *combiner* defines the function that combines two values that have been produced by the *accumulator* function.

Assuming the preceding program, the following statement computes the product of the elements in myList by use of a parallel stream:

```
int parallelProduct = myList.parallelStream().reduce(1,
                                              (a,b) -> a*b,
                                              (a,b) -> a*b);
```

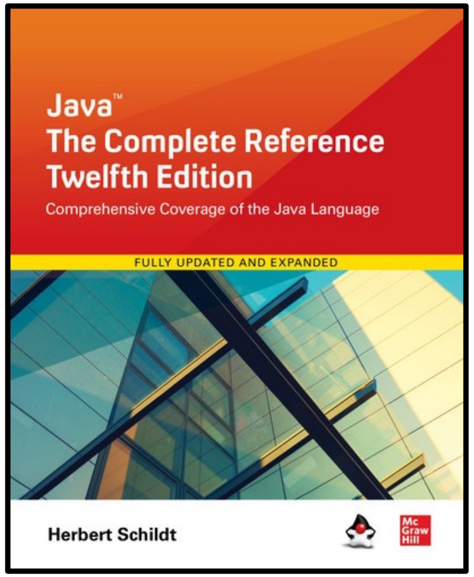



Herb Schildt

As you can see, in this example, both the accumulator and combiner perform the same function. However, there are cases in which the actions of the accumulator must differ from those of the combiner.

For example, consider the following program. Here, myList contains a list of double values. It then uses the combiner version of reduce( ) to compute the product of the *square roots* of each element in the list.

```
import java.util.*;
import java.util.stream.*;

public class StreamDemo3 {

  public static void main(String[] args) {

    // This is now a list of double values
    ArrayList<Double> myList = new ArrayList<>();

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    double productOfSqrRoots = myList.parallelStream().reduce(1.0,
                                                   (a,b) -> a * Math.sqrt(b),
                                                   (a,b) -> a*b);

    System.out.println("Product of square roots: " + productOfSqrRoots);
  }
}
```



Herb Schildt

Notice that the accumulator function multiplies the square roots of two elements, but the combiner multiplies the partial results.

Thus, the two functions differ. Moreover, for this computation to work correctly, they *must* differ. For example, if you tried to obtain the product of the square roots of the elements by using the following statement, an error would result:

```
// this won't work
double productOfSqrRoots2 = myList.parallelStream().reduce(1.0,
                                          (a,b) -> a * Math.sqrt(b));
```

In this version of reduce( ), the accumulator and the combiner function are one and the same.

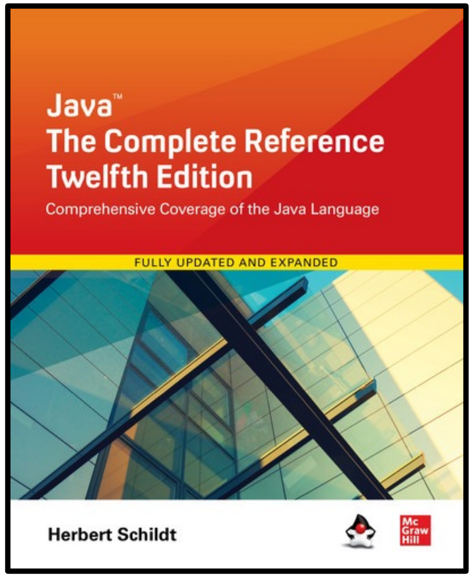
This results in an error because when two partial results are combined, their square roots are multiplied together rather than the partial results, themselves.

As a point of interest, if the stream in the preceding call to reduce( ) had been changed to a sequential stream, then the operation would yield the correct answer because there would have been no need to combine two partial results. The problem occurs when a parallel stream is used.

You can switch a **parallel stream** to **sequential** by calling the **sequential( )** method, which is specified by **BaseStream**. It is shown here:

```
S sequential( )
```

In general, a stream can be switched between **parallel** and **sequential** on an as-needed basis.



Herb Schildt

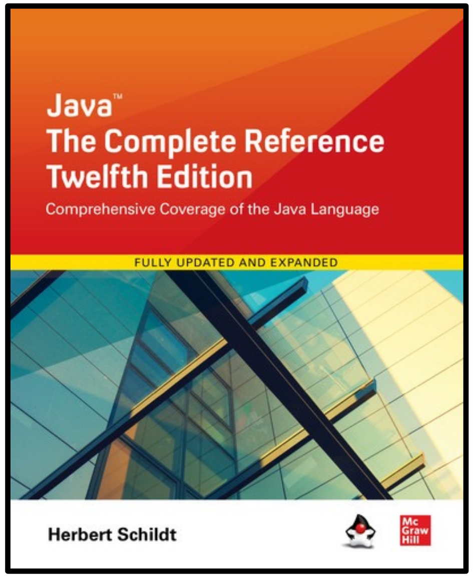
There is one other aspect of a **stream** to keep in mind when using **parallel execution**: the **order** of the elements. **Streams** can be either **ordered** or **unordered**. In general, if the **data source** is **ordered**, then the **stream** will also be **ordered**.

However, when using a **parallel stream**, a **performance boost** can sometimes be obtained by allowing a **stream** to be **unordered**.

When a **parallel stream** is **unordered**, each **partition** of the **stream** can be operated on **independently**, without having to **coordinate** with the others. In cases in which the **order** of the operations does not matter, it is possible to specify **unordered** behavior by calling the **unordered( )** method, shown here:

```
S unordered( )
```

One other point: the **forEach( )** method may not **preserve** the **ordering** of a **parallel stream**. If you want to perform an operation on each element in a **parallel stream** while **preserving** the **order**, consider using **forEachOrdered( )**. It is used just like **forEach( )**.



Herb Schildt

In Scala, this version of the reduce function is called aggregate.

Java

```
<U> U reduce(U identityVal,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)

interface BinaryOperator<T> extends BiFunction<T, T, T>

interface BiFunction<T, U, R>
```

```
def aggregate[S](z: => S)
                (seqop: (S, T) => S,
                 combop: (S, S) => S): S
```

Scala

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Performs a **reduction** on the elements of this **stream**, using the provided **identity**, **accumulation** and **combining** functions. This is equivalent to:

```
    U result = identity;
    for (T element : this stream)
        result = accumulator.apply(result, element)
    return result;
```

but is not constrained to execute **sequentially**.

The **identity** value must be an **identity** for the **combiner** function. This means that for all **u**, **combiner**(**identity**, **u**) is equal to **u**. Additionally, the **combiner** function must be compatible with the **accumulator** function; for all **u** and **t**, the following must hold:

```
combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)
```

This is a terminal operation.

API Note: Many **reductions** using this form can be represented more simply by an explicit combination of **map** and **reduce** operations. The **accumulator** function acts as a fused **mapper** and **accumulator**, which can sometimes be more efficient than separate **mapping** and **reduction**, such as when knowing the previously **reduced** value allows you to avoid some computation.

Type Parameters:
U - The type of the result

Module java.base - **Package** java.util.stream
interface Stream<T>
Type Parameters: T the type of the stream elements

Parameters:

Identity	the identity value for the **combiner** function
accumulator	an associative, non-interfering, stateless function for **incorporating** an additional element into a result
combiner	an associative, non-interfering, stateless function for **combining** two values, which must be compatible with the **accumulator** function

Returns:
the result of the **reduction**

Java

```
def aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

Deprecated - **aggregate** is not relevant for **sequential** collections. Use `` `foldLeft(z)(seqop)` `` instead.

Source: IterableOnce.scala

Scala

```
def aggregate[S](z: => S)(seqop: (S, T) => S, combop: (S, S) => S): S
```

Aggregates the results of applying an **operator** to subsequent elements.

This is a more general form of fold and reduce. It has similar semantics, but does not require the result to be a supertype of the element type. **It traverses the elements in different partitions sequentially, using seqop to update the result, and then applies combop to results from different partitions**. The implementation of this operation may operate on an arbitrary number of collection **partitions**, so **combop** may be invoked arbitrary number of times.

For example, one might want to process some elements and then produce a Set. In this case, **seqop** would process an element and append it to the set, while **combop** would concatenate two sets from different **partitions** together. The initial value **z** would be an empty set.

```
    pc.aggregate(Set[Int]())(_ += process(_), _ ++ _)
```

Another example is calculating geometric mean from a collection of doubles (one would typically require big doubles for this).

Type parameters: S — the type of **accumulated** results

Value parameters:

z	the initial value for the **accumulated** result of the **partition** - this will typically be the neutral element for the **seqop operator** (e.g. **Nil** for list concatenation or **0** for summation) and may be evaluated more than once
seqop	an operator used to **accumulate** results within a **partition**
combop	an **associative** operator used to **combine** results from different **partitions**

Source: ParIterableLike.scala

Non-parallelizable operations

While most parallel collection operations achieve superior performance by executing on several processors, some operations are inherently sequential, and their semantics do not allow them to execute in parallel. Consider the foldLeft method from the Scala collections API:

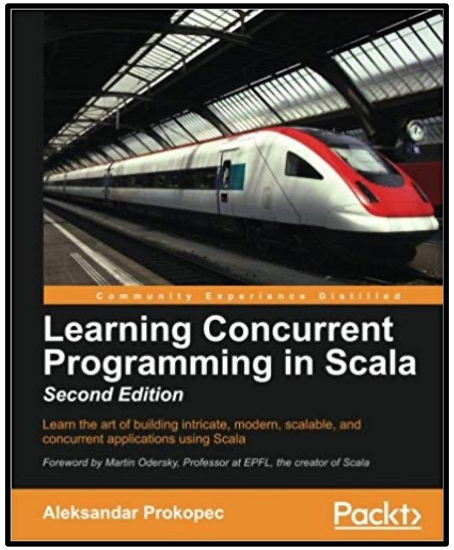
```
def foldLeft[S](z: S)(f: (S, T) => S): S
```

This method visits elements of the collection going from left to right and adds them to the accumulator of type S. The accumulator is initially equal to the zero value z, and is updated with the function f that uses the accumulator and a collection element of type T to produce a new accumulator. For example, given a list of integers List(1, 2, 3), we can compute the sum of its integers with the following expression:

```
List(1, 2, 3).foldLeft(0)((acc, x) => acc + x)
```

This foldLeft method starts by assigning 0 to acc. It then takes the first element in the list 1 and calls the function f to evaluate 0 + 1. The acc accumulator then becomes 1. This process continues until the entire list of elements is visited, and the foldLeft method eventually returns the result 6. In this example, the S type of the accumulator is set to the Int type. In general, the accumulator can have any type. When converting a list of elements to a string, the zero value is an empty string and the function f concatenates a string and a number.

The crucial property of the foldLeft operation is that it traverses the elements of the list by going from left to right. This is reflected in the type of the function f; it accepts an accumulator of type S and a list value of type T. The function f cannot take two values of the accumulator type S and merge them into a new accumulator of type S. As a consequence, computing the accumulator cannot be implemented in parallel; the foldLeft method cannot merge two accumulators from two different processors.



Aleksandar Prokopec
@alexprokopec

We can confirm this by running the following program:

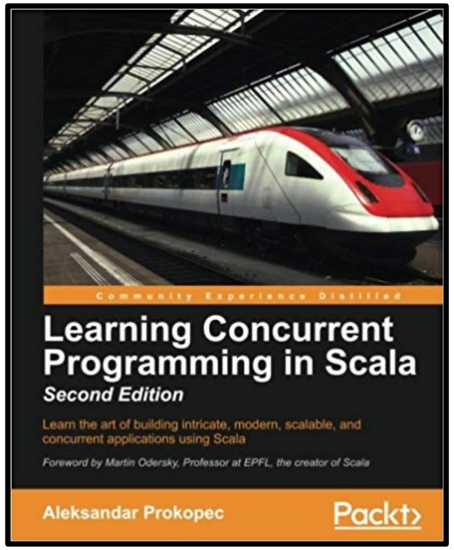
```scala
object ParNonParallelizableOperations extends App {
  import scala.collection._
  import scala.concurrent.ExecutionContext.Implicits.global
  import ParHtmlSpecSearch.getHtmlSpec

  getHtmlSpec() foreach { case specDoc =>
    def allMatches(d: GenSeq[String]) = warmedTimed() {
      val results =
        d.foldLeft("")((acc, line) =>  // Note: must use "aggregate" instead of "foldLeft"!
          if (line.matches(".*TEXTAREA.*")) s"$acc\n$line" else acc)
    }

    val seqtime = allMatches(specDoc)
    log(s"Sequential time - $seqtime ms")

    val partime = allMatches(specDoc.par)
    log(s"Parallel time   - $partime ms")
  }
}
```

Symbol GenSeq is deprecated. Gen* collection types have been removed – 2.13.0

In the preceding program, we use the **getHtmlSpec** method introduced earlier to obtain the lines of the **HTML specification**.

We install a **callback** using the **foreach** call to process the **HTML specification** once it arrives; the **allMatches** method calls the **foldLeft** operation to **accumulate** the lines of the **specification** that contain the **TEXTAREA** string.

Running the program reveals that both the sequential and parallel foldLeft operations take 5.6 milliseconds.



Aleksandar Prokopec
@alexprokopec

Although the key code on this slide is just the bit highlighted in yellow, to help you understand the rest of the code (if you are interested), the next slide covers functions `getHtmlSpec()` and `warmedTimed()`, which were introduced elsewhere in the book.

```scala
object ParHtmlSpecSearch extends App {

  import scala.concurrent._
  import ExecutionContext.Implicits.global
  import scala.collection._
  import scala.io.Source

  def getHtmlSpec() = Future {
    val specSrc: Source = Source.fromURL(
      "http://www.w3.org/MarkUp/html-spec/html-spec.txt")
    try specSrc.getLines.toArray finally specSrc.close()
  }

  getHtmlSpec() foreach { case specDoc =>
    log(s"Download complete!")

    def search(d: GenSeq[String]) = warmedTimed() {
      d.indexWhere(line => line.matches(".*TEXTAREA.*"))
    }

    val seqtime = search(specDoc)
    log(s"Sequential time $seqtime ms")

    val partime = search(specDoc.par)
    log(s"Parallel time $partime ms")
  }

}
```

Symbol GenSeq is deprecated. Gen* collection types have been removed – 2.13.0

```scala
@volatile var dummy: Any = _
def timed[T](body: =>T): Double = {
  val start = System.nanoTime
  dummy = body
  val end = System.nanoTime
  ((end - start) / 1000) / 1000.0
}
```

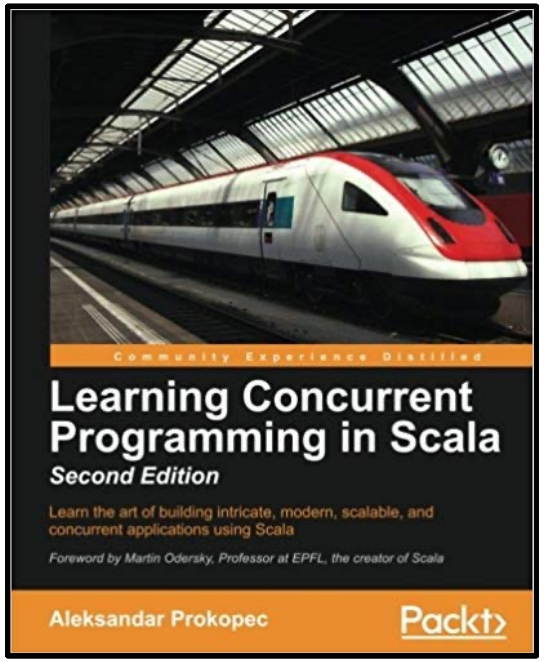
```scala
def warmedTimed[T](n: Int = 200)(body: =>T): Double = {
  for (_ <- 0 until n) body
  timed(body)
}
```

This method runs the block of code n times before measuring its running time. We set the default value for the n variable to 200; although there is no way to be sure that the JVM will reach a steady state after executing the block of code 200 times, this is a reasonable default.

...programs running on the JVM are usually slow immediately after they start, and eventually reach their optimal performance. Once this happens, we say that the JVM reached its steady state. When evaluating the performance on the JVM, we are usually interested in the steady state; most programs run long enough to achieve it.

To witness this effect, assume that you want to find out what the TEXTAREA tag means in HTML. You write the program that downloads the HTML specification and searches for the first occurrence of the TEXTAREA string.

...implement the getHtmlSpec method, which starts an asynchronous computation to download the HTML specification and returns a future value with the lines of the HTML specification. You then install a callback; once the HTML specification is available, you can call the indexWhere method on the lines to find the line that matches the regular expression .*TEXTAREA.*



To specify how the accumulators produced by different processors should be merged together, we need to use the aggregate method.

The aggregate method is similar to the foldLeft operation, but it does not specify that the elements are traversed from left to right. Instead, it only specifies that subsets of elements are visited going from left to right; each of these subsets can produce a separate accumulator. The aggregate method takes an additional function of type (S, S) => S, which is used to merge multiple accumulators.

```
def aggregate[S](z: => S)(seqop: (S, T) => S, combop: (S, S) => S): S
```

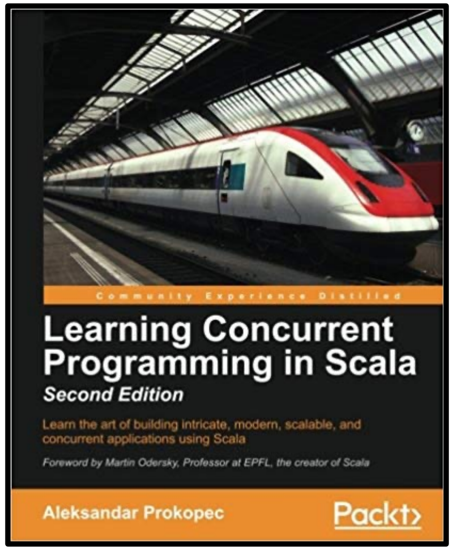
```
d.aggregate("")
           ((acc, line) =>   if (line.matches(".*TEXTAREA.*")) s"$acc\n$line" else acc,
            (acc1, acc2) => acc1 + acc2 )
```

Running the example again shows the difference between the sequential and parallel versions of the program; the parallel aggregate method takes 1.4 milliseconds to complete on our machine.

When doing these kinds of reduction operation in parallel, we can alternatively use the reduce or fold methods, which do not guarantee going from left to right. The aggregate method is more expressive, as it allows the accumulator type to be different from the type of the elements in the collection.

TIP Use the aggregate method to execute parallel reduction operations.

Other inherently sequential operations include foldRight, reduceLeft, reduceRight, reduceLeftOption, reduceRightOption, scanLeft, scanRight, and methods that produce non-parallelizable collections such as the toList method.



Aleksandar Prokopec
@alexprokopec

Commutative and associative operators

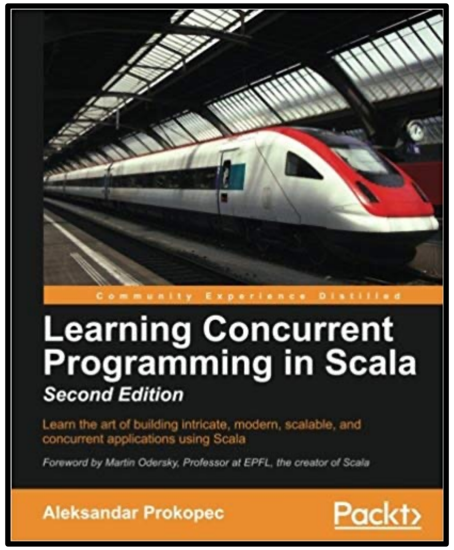
Parallel collection operations such as reduce, fold, aggregate, and scan take binary operators as part of their input. A binary operator is a function op that takes two arguments, a and b. We can say that the binary operator op is commutative if changing the order of its arguments returns the same result, that is, op(a, b) == op(b, a). For example, adding two numbers together is a commutative operation. Concatenating two strings is not a commutative operation; we get different strings depending on the concatenation order.

Binary operators for the parallel reduce, fold, aggregate, and scan operations never need to be commutative. Parallel collection operations always respect the relative order of the elements when applying binary operators, provided that the underlying collections have any ordering. Elements in sequence collections, such as ArrayBuffer collections, are always ordered. Other collection types can order their elements but are not required to do so.

In the following example, we can concatenate the strings inside an ArrayBuffer collection into one long string by using the sequential reduceLeft operation and the parallel reduce operation. We then convert the ArrayBuffer collection into a set, which does not have an ordering:

Gen* collection types have been removed – 2.13.0

```
object ParNonCommutativeOperator extends App {
  import scala.collection._

  val doc = mutable.ArrayBuffer.tabulate(20)(i => s"Page $i, ")
  def test(doc: GenIterable[String]) {
    val seqtext = doc.seq.reduceLeft(_ + _)
    val partext = doc.par.reduce(_ + _)
    log(s"Sequential result - $seqtext\n")
    log(s"Parallel result   - $partext\n")
  }
  test(doc)
  test(doc.toSet)
}
```



Aleksandar Prokopec
@alexprokopec

We can see that the string is concatenated correctly when the parallel reduce operation is invoked on a parallel array, but the order of the pages is mangled both for the reduceLeft and reduce operations when invoked on a set; the default Scala set implementation does not order the elements.

NOTE

Binary operators used in parallel operations do not need to be commutative.

An op binary operator is associative if applying op consecutively to a sequence of values a, b, and c gives the same result regardless of the order in which the operator is applied, that is, op(a, op(b, c)) == op(op(a, b), c). Adding two numbers together or computing the larger of the two numbers is an associative operation. Subtraction is not associative, as 1 - (2 - 3) is different from (1 - 2) - 3.

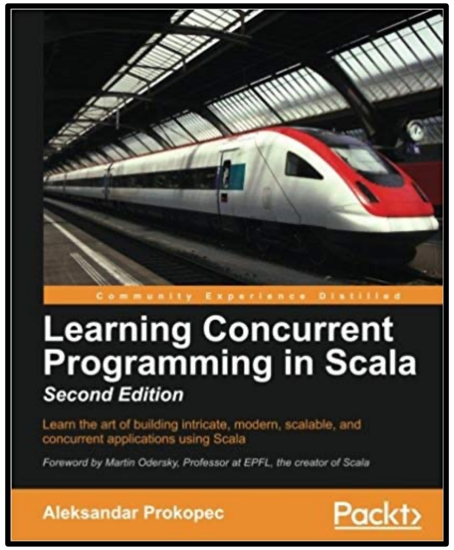
Parallel collection operations usually require associative binary operators. While using subtraction with the reduceLeft operation means that all the numbers in the collection should be subtracted from the first number, using subtraction in the reduce, fold, or scan methods gives nondeterministic and incorrect results, as illustrated by the following code snippet:

```
object ParNonAssociativeOperator extends App {
  import scala.collection._
  def test(doc: GenIterable[Int]) {
    val seqtext = doc.seq.reduceLeft(_ - _)
    val partext = doc.par.reduce(_ - _)
    log(s"Sequential result - $seqtext\n")
    log(s"Parallel result   - $partext\n")
  }
  test(0 until 30)
}
```

While the reduceLeft operation consistently returns -435, the reduce operation returns meaningless results at random.

TIP

Make sure that binary operators used in parallel operations are associative.



Aleksandar Prokopec
@alexprokopec

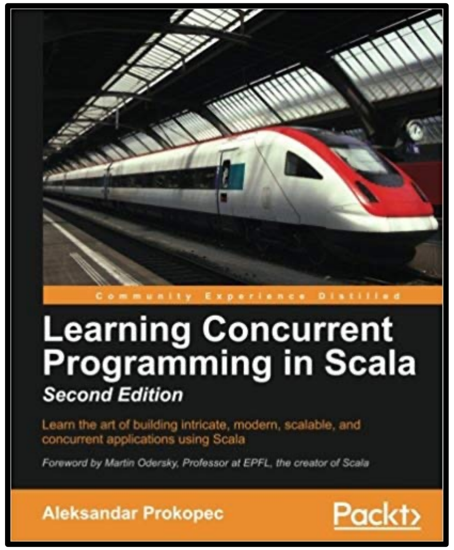
Parallel operations such as aggregate require the multiple binary operators, sop and cop:

```
def aggregate[S](z: => S)(sop: (S, T) => S, cop: (S, S) => S): S
```

The sop operator is of the same type as the operator required by the reduceLeft operation. It takes an accumulator and the collection element. The sop operator is used to fold elements within a subset assigned to a specific processor.

The cop operator is used to merge the subsets together and is of the same type as the operators for reduce and fold.

The aggregate operation requires that cop is associative and that z is the zero element for the accumulator, that is, cop(z, a) == a. Additionally, the sop and cop operators must give the same result irrespective of the order in which element subsets are assigned to processors, that is, cop(sop(z, a), sop(z, b)) == cop(z, sop(sop(z, a), b)).



Aleksandar Prokopec
@alexprokopec

That’s all.

I hope you found it useful.

@philip_schwarz