TABLE 4.1 Common functional interfaces

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
`Supplier<T>`	0	`T`	`get`
`Consumer<T>`	1 (`T`)	`void`	`accept`
`BiConsumer<T, U>`	2 (`T`, `U`)	`void`	`accept`
`Predicate<T>`	1 (`T`)	`boolean`	`test`
`BiPredicate<T, U>`	2 (`T`, `U`)	`boolean`	`test`
`Function<T, R>`	1 (`T`)	`R`	`apply`
`BiFunction<T, U, R>`	2 (`T`, `U`)	`R`	`apply`
`UnaryOperator<T>`	1 (`T`)	`T`	`apply`
`BinaryOperator<T>`	2 (`T`, `T`)	`T`	`apply`

TABLE 4.2 `Optional` instance methods

Method	When `Optional` Is Empty	When `Optional` Contains a Value
`get()`	Throws an exception	Returns value
`ifPresent(Consumer c)`	Does nothing	Calls `Consumer c` with value
`isPresent()`	Returns `false`	Returns `true`
`orElse(T other)`	Returns `other` parameter	Returns value
`orElseGet(Supplier s)`	Returns result of calling `Supplier`	Returns value
`orElseThrow(Supplier s)`	Throws exception created by calling `Supplier`	Returns value

TABLE 4.3 Intermediate vs. terminal operations

Scenario	For Intermediate Operations?	For Terminal Operations?
Required part of a useful pipeline?	No	Yes
Can exist multiple times in a pipeline?	Yes	No
Return type is a stream type?	Yes	No
Executed upon method call?	No	Yes
Stream valid after call?	Yes	No

TABLE 4.4 Terminal stream operations

Method	What Happens for Infinite Streams	Return Value	Reduction
`allMatch()` `/anyMatch()` `/noneMatch()`	Sometimes terminates	`boolean`	No
`collect()`	Does not terminate	Varies	Yes
`count()`	Does not terminate	`long`	Yes
`findAny()` `/findFirst()`	Terminates	`Optional<T>`	No
`forEach()`	Does not terminate	`void`	No
`min()/max()`	Does not terminate	`Optional<T>`	Yes
`reduce()`	Does not terminate	Varies	Yes

TABLE 4.5 How to print a stream

Option	Works for Infinite Streams?	Destructive to Stream?
`s.forEach(System.out::println);`	No	Yes
`System.out.println(s.collect(Collectors.toList()));`	No	Yes
`s.peek(System.out::println).count();`	No	No
`s.limit(5).forEach(System.out::println);`	Yes	Yes

TABLE 4.6 Mapping methods between types of streams

Source Stream Class	To Create `Stream`	To Create `DoubleStream`	To Create `IntStream`	To Create `LongStream`
`Stream`	`map`	`mapToDouble`	`mapToInt`	`mapToLong`
`DoubleStream`	`mapToObj`	`map`	`mapToInt`	`mapToLong`
`IntStream`	`mapToObj`	`mapToDouble`	`map`	`mapToLong`
`LongStream`	`mapToObj`	`mapToDouble`	`mapToInt`	`map`

TABLE 4.7 Function parameters when mapping between types of streams

Source Stream Class	To Create Stream	To Create DoubleStream	To Create IntStream	To Create LongStream
Stream	Function	ToDoubleFunction	ToIntFunction	ToLongFunction
DoubleStream	Double Function	DoubleUnary Operator	DoubleToInt Function	DoubleToLong Function
IntStream	IntFunction	IntToDouble Function	IntUnary Operator	IntToLong Function
LongStream	Long Function	LongToDouble Function	LongToInt Function	LongUnary Operator

TABLE 4.8 Optional types for primitives

	OptionalDouble	OptionalInt	OptionalLong
Getting as a primitive	getAsDouble()	getAsInt()	getAsLong()
orElseGet() parameter type	DoubleSupplier	IntSupplier	LongSupplier
Return type of max()	OptionalDouble	OptionalInt	OptionalLong
Return type of sum()	double	int	long
Return type of avg()	OptionalDouble	OptionalDouble	OptionalDouble

TABLE 4.9 Common functional interfaces for primitives

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
DoubleSupplier	0	double	getAsDouble
IntSupplier		int	getAsInt
LongSupplier		long	getAsLong
DoubleConsumer	1 (double)	void	accept
IntConsumer	1 (int)		
LongConsumer	1 (long)		
DoublePredicate	1 (double)	boolean	test
IntPredicate	1 (int)		
LongPredicate	1 (long)		

TABLE 4.10 Common functional interfaces for primitives *(continued)*

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
`DoubleFunction<R>`	1 (`double`)	`R`	`apply`
`IntFunction<R>`	1 (`int`)		
`LongFunction<R>`	1 (`long`)		
`DoubleUnaryOperator`	1 (`double`)	`double`	`applyAsDouble`
`IntUnaryOperator`	1 (`int`)	`int`	`applyAsInt`
`LongUnaryOperator`	1 (`long`)	`long`	`applyAsLong`
`DoubleBinaryOperator`	2 (`double, double`)	`double`	`applyAsDouble`
`IntBinaryOperator`	2 (`int, int`)	`int`	`applyAsInt`
`LongBinaryOperator`	2 (`long, long`)	`long`	`applyAsLong`

TABLE 4.10 Primitive-specific functional interfaces

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
`ToDoubleFunction<T>`	1 (`T`)	`double`	`applyAsDouble`
`ToIntFunction<T>`		`int`	`applyAsInt`
`ToLongFunction<T>`		`long`	`applyAsLong`
`ToDoubleBiFunction<T, U>`	2 (`T, U`)	`double`	`applyAsDouble`
`ToIntBiFunction<T, U>`		`int`	`applyAsInt`
`ToLongBiFunction<T, U>`		`long`	`applyAsLong`

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
`DoubleToIntFunction`	1 (`double`)	`int`	`applyAsInt`
`DoubleToLongFunction`	1 (`double`)	`long`	`applyAsLong`
`IntToDoubleFunction`	1 (`int`)	`double`	`applyAsDouble`
`IntToLongFunction`	1 (`int`)	`long`	`applyAsLong`
`LongToDoubleFunction`	1 (`long`)	`double`	`applyAsDouble`
`LongToIntFunction`	1 (`long`)	`int`	`applyAsInt`
`ObjDoubleConsumer<T>`	2 (`T, double`)	`void`	`accept`
`ObjIntConsumer<T>`	2 (`T, int`)		
`ObjLongConsumer<T>`	2 (`T, long`)		

TABLE 4.11 Examples of grouping/partitioning collectors

Collector	Description	Return Value When Passed to `collect`

`averagingDouble(ToDoubleFunction f)`
`averagingInt(ToIntFunction f)`
`averagingLong(ToLongFunction f)`
	Calculates the average for our three core primitive types	`Double`

`counting()`	Counts number of elements	`Long`

`groupingBy(Function f)`
`groupingBy(Function f, Collector dc)`
`groupingBy(Function f, Supplier s, Collector dc)`
	Creates a map grouping by the specified function with the optional type and optional downstream collector	`Map<K, List<T>>`

`joining()` `joining(CharSequence cs)`	Creates a single `String` using `cs` as a delimiter between elements if one is specified	`String`
`maxBy(Comparator c)` `minBy(Comparator c)`	Finds the largest/smallest elements	`Optional<T>`
`mapping(Function f, Collector dc)`	Adds another level of collectors	`Collector`
`partitioningBy(Predicate p)` `partitioningBy(Predicate p, Collector dc)`	Creates a map grouping by the specified predicate with the optional further downstream collector	`Map<Boolean, List<T>>`

Collector	Description	Return Value When Passed to `collect`

`summarizingDouble(ToDoubleFunction f)`
`summarizingInt(ToIntFunction f)`
`summarizingLong(ToLongFunction f)`
	Calculates average, min, max, and so on	`DoubleSummaryStatistics`
		`IntSummaryStatistics`
		`LongSummaryStatistics`

`summingDouble(ToDoubleFunction f)`
`summingInt(ToIntFunction f)`
`summingLong(ToLongFunction f)`
	Calculates the sum for our three core primitive types	`Double`
		`Integer`
		`Long`

`toList()` `toSet()`	Creates an arbitrary type of list or set	`List` `Set`
`toCollection(Supplier s)`	Creates a `Collection` of the specified type	`Collection`

`toMap(Function k, Function v)`
`toMap(Function k, Function v, BinaryOperator m)`
`toMap(Function k, Function v, BinaryOperator m, Supplier s)`
	Creates a map using functions to map the keys, values, an optional merge function, and an optional type	`Map`