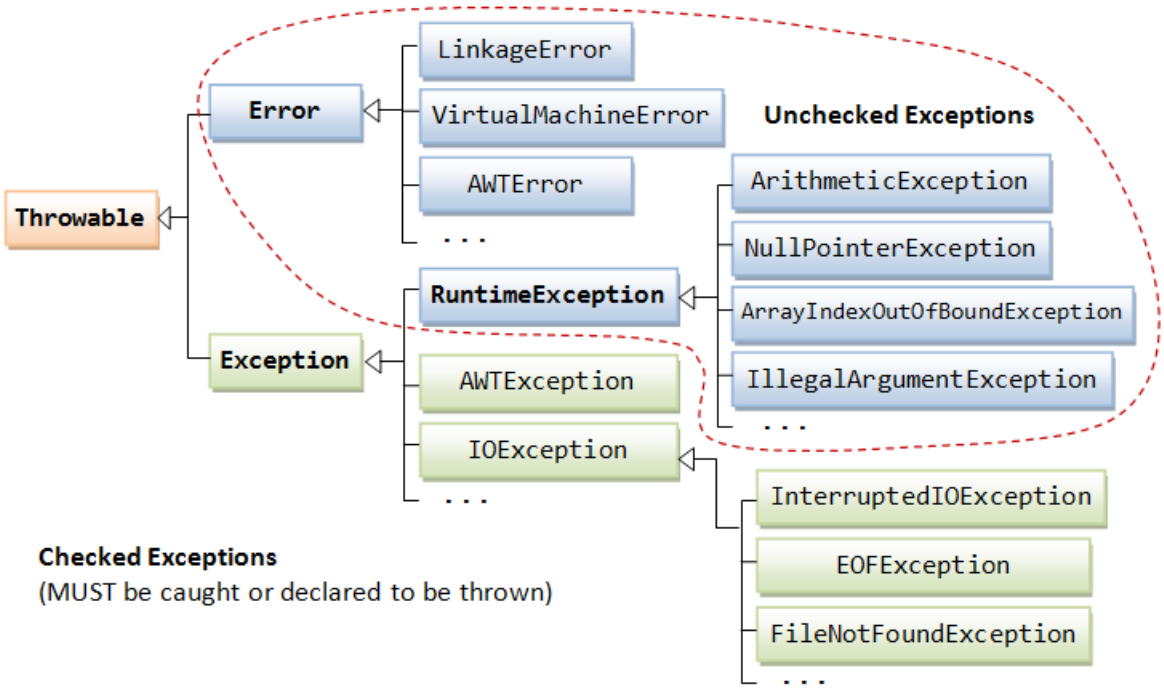
Property	Overloaded method	Overridden method
Access level	Can Change	Must same or less restrictive
Argument list	Must change	Must not change
Exceptions	Can change	Can throw new or broader runtime exceptions
Return type	Can change	Must not change except for covariant returns
Invocation	Reference type determines which version	Object type determines which version

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides



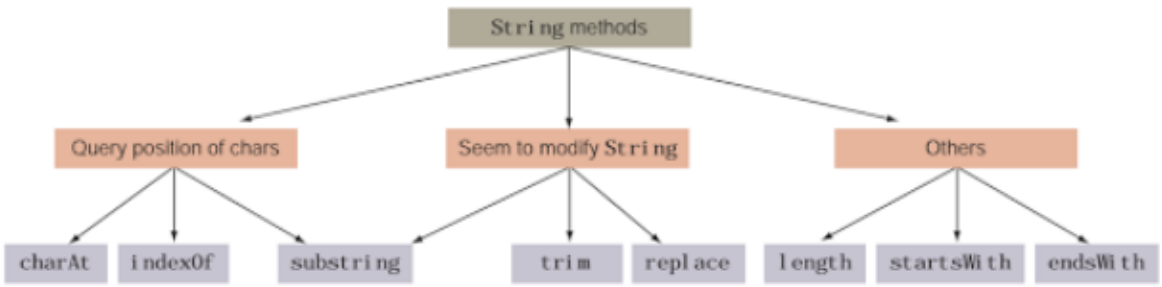


Figure 4.8 Categorization of the String methods

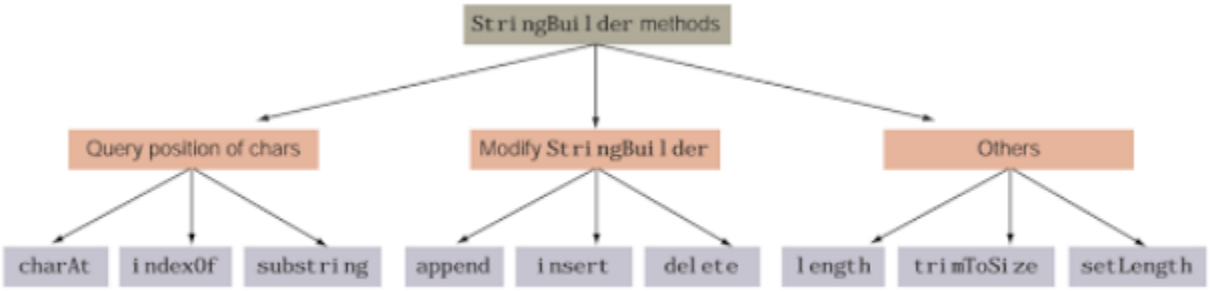


Figure 4.14 Categorization of `StringBuilder` methods

Operator Precedence

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

TABLE 3.7 Java Collections Framework types

Type	Can contain duplicate elements?	Elements ordered?	Has keys and values?	Must add/remove in specific order?
`List`	Yes	Yes (by index)	No	No
`Map`	Yes (for values)	No	Yes	No
`Queue`	Yes	Yes (retrieved in defined order)	No	Yes
`Set`	No	No	No	No

Type	Java Collections Framework interface	Sorted?	Calls `hashCode`?	Calls `compareTo`?
`ArrayList`	`List`	No	No	No
`ArrayDeque`	`Queue`	No	No	No
`HashMap`	`Map`	No	Yes	No
`HashSet`	`Set`	No	Yes	No
`Hashtable`	`Map`	No	Yes	No
`LinkedList`	`List`, `Queue`	No	No	No
`Stack`	`List`	No	No	No
`TreeMap`	`Map`	Yes	No	Yes
`TreeSet`	`Set`	Yes	No	Yes
`Vector`	`List`	No	No	No

TABLE 3.10 Comparison of `Comparable` and `Comparator`

Difference	`Comparable`	`Comparator`
Package name	`java.lang`	`java.util`
Interface must be implemented by class comparing?	Yes	No
Method name in interface	`compareTo`	`compare`
Number of parameters	1	2
Common to declare using a lambda	No	Yes

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 4.4 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 2–1 Functional Interfaces Used in the Stream API

Functional Interface	Parameter Types	Return Type	Description
`Supplier<T>`	None	`T`	Supplies a value of type `T`
`Consumer<T>`	`T`	`void`	Consumes a value of type `T`
`BiConsumer<T, U>`	`T, U`	`void`	Consumes values of types `T` and `U`
`Predicate<T>`	`T`	`boolean`	A Boolean-valued function

`ToIntFunction<T>`
`ToLongFunction<T>`
`ToDoubleFunction<T>`
	`T`	`int`
		`long`
		`double`
			An `int`-, `long`-, or `double`-valued function

`IntFunction<R>`
`LongFunction<R>`
`DoubleFunction<R>`
	`int`
	`long`
	`double`
		`R`	A function with argument of type `int`, `long`, or `double`

`Function<T, R>`	`T`	`R`	A function with argument of type `T`
`BiFunction<T, U, R>`	`T, U`	`R`	A function with arguments of types `T` and `U`
`UnaryOperator<T>`	`T`	`T`	A unary operator on the type `T`
`BinaryOperator<T>`	`T, T`	`T`	A binary operator on the type `T`

Table 3–2 Functional Interfaces for Primitive Types
p, *q* is int, long, double;
P, *Q* is Int, Long, Double

Functional Interface	Parameter Types	Return Type	Abstract Method Name
BooleanSupplier	none	boolean	getAsBoolean
*P*Supplier	none	*p*	getAs*P*
*P*Consumer	*p*	void	accept
Obj*P*Consumer<T>	T, *p*	void	accept
*P*Function<T>	*p*	T	apply
*P*To*Q*Function	*p*	*q*	applyAs*Q*
To*P*Function<T>	T	*p*	applyAs*P*
To*P*BiFunction<T, U>	T, U	*p*	applyAs*P*
*P*UnaryOperator	*p*	*p*	applyAs*P*
*P*BinaryOperator	*p*, *p*	*p*	applyAs*P*
*P*Predicate	*p*	boolean	test

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)		

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)		

–

memorize the contents of Table 6.2 and Table 6.3 before the exam

TABLE 6.2 OCP checked exceptions

Exception	Used when	Checked or unchecked?	Where to find more details
`java.text.ParseException`	Converting a `String` to a number.	Checked	Chapter 5

`java.io.IOException`
`java.io.FileNotFoundException`
`java.io.NotSerializableException`
	Dealing with IO and NIO.2 issues. `IOException` is the parent class. There are a number of subclasses. You can assume any `java.io` exception is checked.	Checked	Chapter 9

`java.sql.SQLException`	Dealing with database issues. `SQLException` is the parent class. Again, you can assume any `java.sql` exception is checked.	Checked	Chapter 10

TABLE 6.3 OCP runtime exceptions

Exception	Used when	Checked or unchecked?	Where to find more details
`java.lang.ArrayStoreException`	Trying to store the wrong data type in an array.	Unchecked	Chapter 3
`java.time.DateTimeException`	Receiving an invalid format string for a date.	Unchecked	Chapter 3
`java.util.MissingResourceException`	Trying to access a key or resource bundle that does not exist.	Unchecked	Chapter 5
`java.lang.IllegalStateException` `java.lang.UnsupportedOperationException`	Attempting to run an invalid operation in collections and concurrency.	Unchecked	Chapters 3 and 7

TABLE 6.4 Legal vs. illegal configurations with a traditional `try` statement

	0 `finally` **blocks**	**1** `finally` **block**	**2 or more** `finally` **blocks**
0 `catch` **blocks**	Not legal	Legal	Not legal
1 or more `catch` **blocks**	Legal	Legal	Not legal

TABLE 6.5 Legal vs. illegal configurations with a try-with-resources statement

	0 `finally` **blocks**	**1** `finally` **block**	**2 or more** `finally` **blocks**
0 `catch` **blocks**	Legal	Legal	Not legal
1 or more `catch` **blocks**	Legal	Legal	Not legal

TABLE 7.2 `ExecutorService` methods

Method Name	**Description**
`void execute(Runnable command)`	Executes a `Runnable` task at some point in the future
`Future<?> submit(Runnable task)`	Executes a `Runnable` task at some point in the future and returns a `Future` representing the task
`<T> Future<T> submit(Callable<T> task)`	Executes a `Callable` task at some point in the future and returns a `Future` representing the pending results of the task
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException`	Executes the given tasks, synchronously returning the results of all tasks as a `Collection` of `Future` objects, in the same order they were in the original collection
`<T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException`	Executes the given tasks, synchronously returning the result of one of finished tasks, cancelling any unfinished tasks

-

TABLE 7.3 Future methods

Method Name	Description
`boolean isDone()`	Returns `true` if the task was completed, threw an exception, or was cancelled.
`boolean isCancelled()`	Returns `true` if the task was cancelled before it completely normally.
`boolean cancel()`	Attempts to cancel execution of the task.
`V get()`	Retrieves the result of a task, waiting endlessly if it is not yet available.
`V get(long timeout, TimeUnit unit)`	Retrieves the result of a task, waiting the specified amount of time. If the result is not ready by the time the timeout is reached, a checked `TimeoutException` will be thrown.

TABLE 7.5 ScheduledExecutorService methods

Method Name	Description
`schedule(Callable<V> callable, long delay, TimeUnit unit)`	Creates and executes a `Callable` task after the given delay
`schedule(Runnable command, long delay, TimeUnit unit)`	Creates and executes a `Runnable` task after the given delay
`scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`	Creates and executes a `Runnable` task after the given initial delay, creating a new task every period value that passes.
`scheduleAtFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`	Creates and executes a `Runnable` task after the given initial delay and subsequently with the given delay between the termination of one execution and the commencement of the next

TABLE 7.6 Executors methods

Method Name	Return Type	Description
`newSingleThreadExecutor()`	`ExecutorService`	Creates a single-threaded executor that uses a single worker thread operating off an unbounded queue. Results are processed sequentially in the order in which they are submitted.
`newSingleThreadScheduledExecutor()`	`ScheduledExecutorService`	Creates a single-threaded executor that can schedule commands to run after a given delay or to execute periodically.
`newCachedThreadPool()`	`ExecutorService`	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
`newFixedThreadPool(int nThreads)`	`ExecutorService`	Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
`newScheduledThreadPool(int nThreads)`	`ScheduledExecutorService`	Creates a thread pool that can schedule commands to run after a given delay or to execute periodically.

TABLE 7.7 Atomic classes

Class Name	Description
`AtomicBoolean`	A `boolean` value that may be updated atomically
`AtomicInteger`	An `int` value that may be updated atomically
`AtomicIntegerArray`	An `int` array in which elements may be updated atomically
`AtomicLong`	A `long` value that may be updated atomically
`AtomicLongArray`	A `long` array in which elements may be updated atomically
`AtomicReference`	A generic object reference that may be updated atomically
`AtomicReferenceArray`	An array of generic object references in which elements may be updated atomically

TABLE 7.8 Common atomic methods

Class Name	Description
`get()`	Retrieve the current value
`set()`	Set the given value, equivalent to the assignment = operator
`getAndSet()`	Atomically sets the new value and returns the old value
`incrementAndGet()`	For numeric classes, atomic pre-increment operation equivalent to `++value`
`getAndIncrement()`	For numeric classes, atomic post-increment operation equivalent to `value++`
`decrementAndGet()`	For numeric classes, atomic pre-decrement operation equivalent to `--value`
`getAndDecrement()`	For numeric classes, atomic post-decrement operation equivalent to `value--`

TABLE 7.9 Concurrent collection classes

Class Name	Java Collections Framework Interface	Elements Ordered?	Sorted?	Blocking?
`ConcurrentHashMap`	`ConcurrentMap`	No	No	No
`ConcurrentLinkedDeque`	`Deque`	Yes	No	No
`ConcurrentLinkedQueue`	`Queue`	Yes	No	No

`ConcurrentSkipListMap`	`ConcurrentMap`
	`SortedMap`
	`NavigableMap`
		Yes	Yes	No

`ConcurrentSkipListSet`	`SortedSet` `NavigableSet`	Yes	Yes	No
`CopyOnWriteArrayList`	`List`	Yes	No	No
`CopyOnWriteArraySet`	`Set`	No	No	No
`LinkedBlockingDeque`	`BlockingQueue` `BlockingDeque`	Yes	No	Yes
`LinkedBlockingQueue`	`BlockingQueue`	Yes	No	Yes

TABLE 10.2 ResultSet type options

ResultSet Type	Can Go Backward	See Latest Data from Database Table	Supported by Most Drivers
ResultSet.TYPE_FORWARD_ONLY	No	No	Yes
ResultSet.TYPE_SCROLL_INSENSITIVE	Yes	No	Yes
ResultSet.TYPE_SCROLL_SENSITIVE	Yes	Yes	No

TABLE 10.3 ResultSet concurrency mode options

ResultSet Type	Can Read Data	Can Update Data	Supported by All Drivers
ResultSet.CONCUR_READ_ONLY	Yes	Yes	No
ResultSet.CONCUR_UPDATABLE	Yes	No	Yes

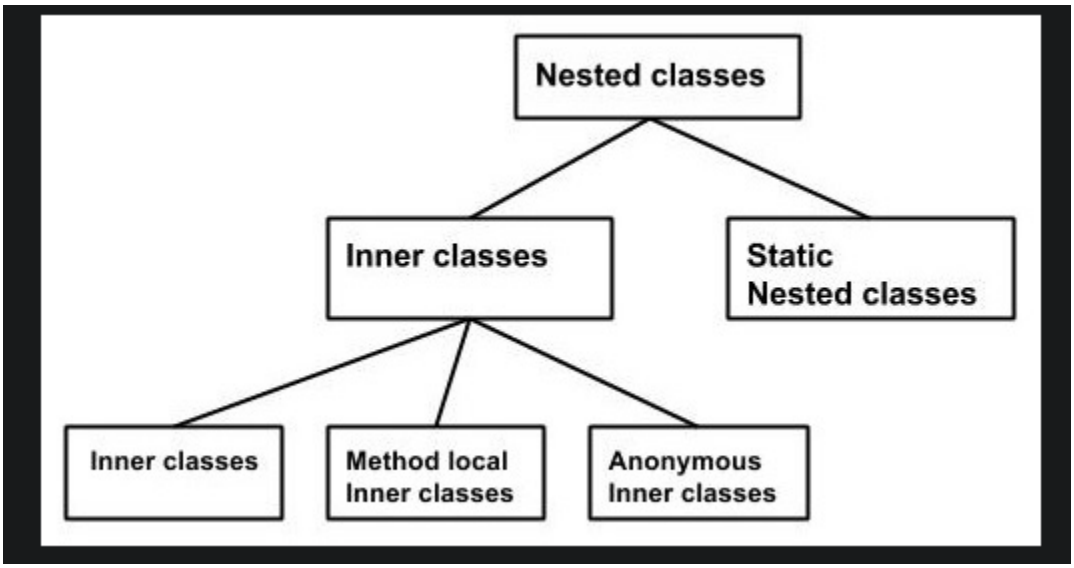


TABLE 10.5 Return types of executes

Method	Return Type	What Is Returned for SELECT	What Is Returned for DELETE/INSERT/UPDATE
`stmt.execute()`	`boolean`	`true`	`false`
`stmt.executeQuery()`	`ResultSet`	The rows and columns returned	n/a
`stmt.executeUpdate()`	`int`	n/a	Number of rows added/changed/removed

TABLE 10.4 SQL runnable by execute method

Method	DELETE	INSERT	SELECT	UPDATE
`stmt.execute()`	Yes	Yes	Yes	Yes
`stmt.executeQuery()`	No	No	Yes	No
`stmt.executeUpdate()`	Yes	Yes	No	Yes

TABLE 10.6 `ResultSet` get methods

Method Name	Return Type	Example Database Type
`getBoolean`	`boolean`	`BOOLEAN`
`getDate`	`java.sql.Date`	`DATE`
`getDouble`	`double`	`DOUBLE`
`getInt`	`int`	`INTEGER`
`getLong`	`long`	`BIGINT`
`getObject`	`Object`	Any type
`getString`	`String`	`CHAR, VARCHAR`
`getTime`	`java.sql.Time`	`TIME`
`getTimeStamp`	`java.sql.TimeStamp`	`TIMESTAMP`

TABLE 10.7 JDBC date and time types

JDBC Type	Java 8 Type	Contains
`java.sql.Date`	`java.time.LocalDate`	Date only
`java.sql.Time`	`java.time.LocalTime`	Time only
`java.sql.TimeStamp`	`java.time.LocalDateTime`	Both date and time

TABLE 10.8 Navigating a `ResultSet`

Method	Description	Requires Scrollable `ResultSet`
`boolean absolute(int rowNum)`	Move cursor to the specified row number	Yes
`void afterLast()`	Move cursor to a location immediately after the last row	Yes
`void beforeFirst()`	Move cursor to a location immediately before the first row	Yes
`boolean first()`	Move cursor to the first row	Yes
`boolean last()`	Move cursor to the last row	Yes
`boolean next()`	Move cursor one row forward	No
`boolean previous()`	Move cursor one row backward	Yes
`boolean relative(int rowNum)`	Move cursor forward or backward the specified number of rows	Yes

FIGURE 7.2 ExecutorService life cycle