Session 4 : Collections, Generics, Streams

sourcemind

Java Collections

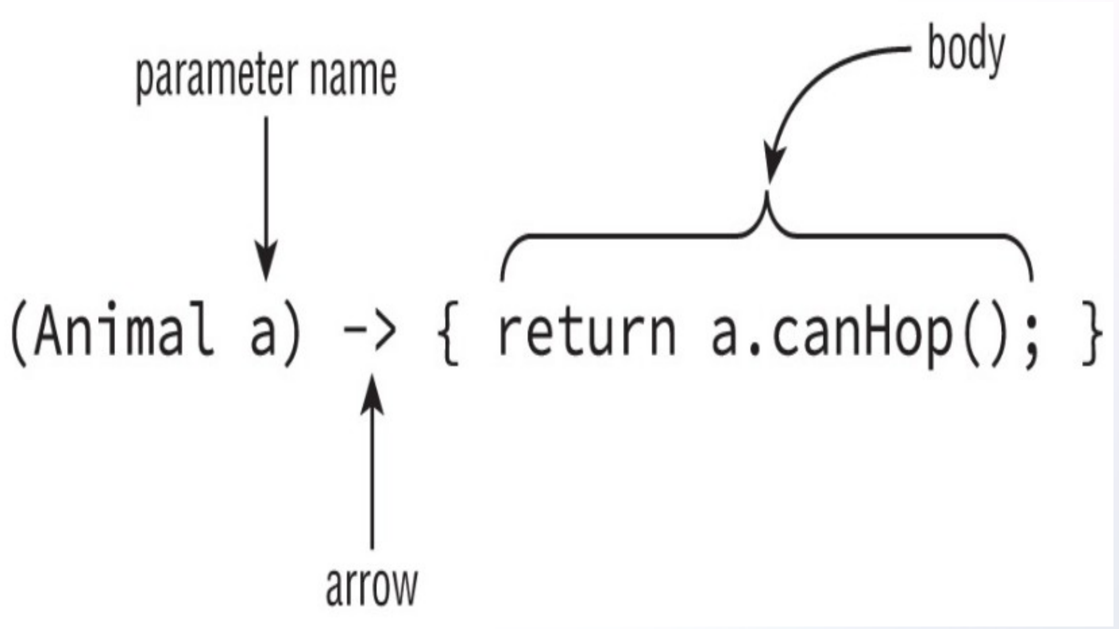
- A collection is a group of objects contained in a single object
- The Java **Collections Framework** is a set of classes in **java.util** for storing collections

Lambdas and Functional Interfaces

sourcemind

Lambdas

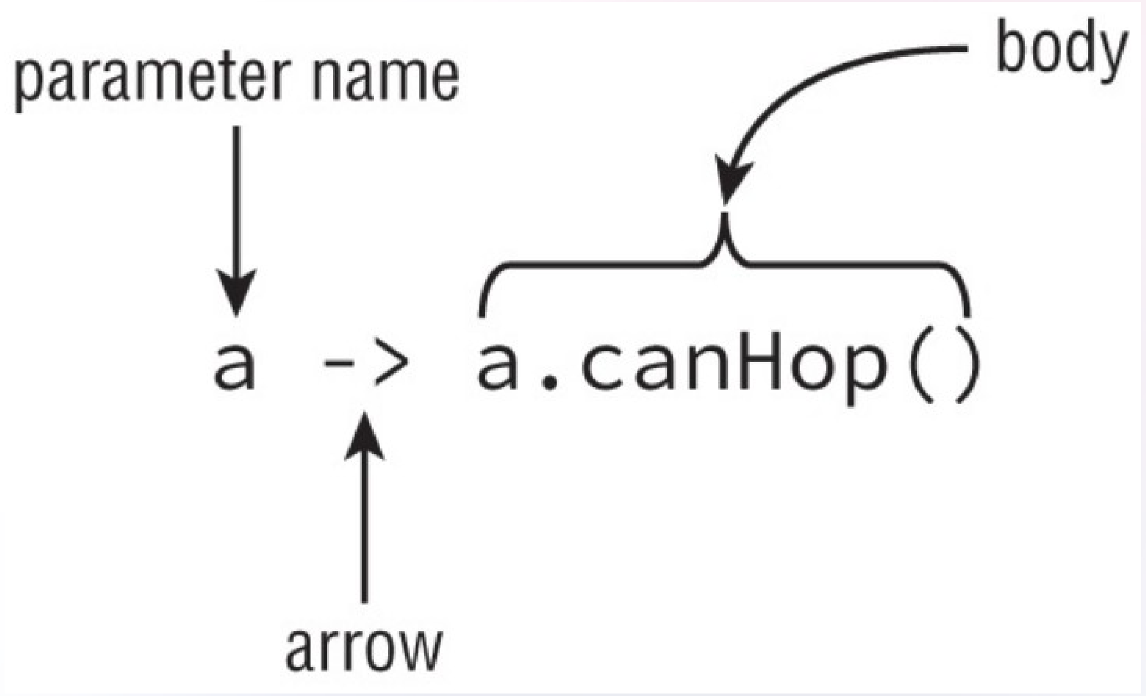
- A **lambda expression** is a block of code that gets passed around. You can think of a lambda expression as an unnamed method.



sourcemind

Optional parts

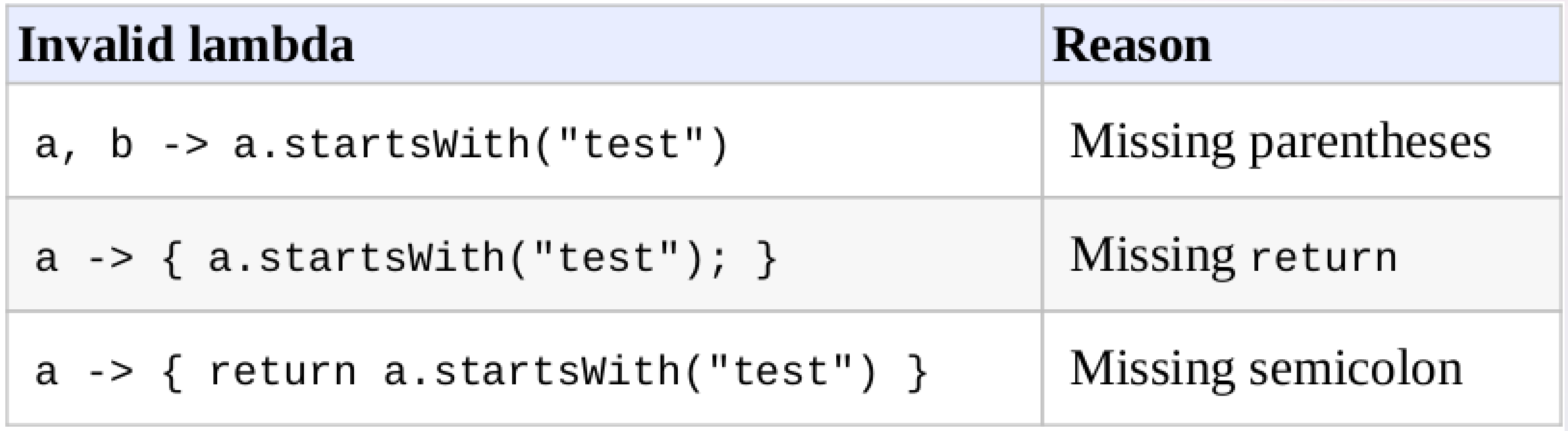
- The parentheses can be omitted only if there is a single parameter and its type is not explicitly stated
- We can omit braces when we have only a single statement
- Java doesn’t require you to type return or use a semicolon when no braces are used. This special shortcut doesn’t work when we have two or more statements



Valid lambdas

Lambda	# parameters
`() -> true`	0
`a -> a.startsWith("test")`	1
`(String a) -> a.startsWith("test")`	1
`(a, b) -> a.startsWith("test")`	2
`(String a, String b) -> a.startsWith("test")`	2

Invalid lambdas



Invalid lambda	Reason
`a, b -> a.startsWith("test")`	Missing parentheses
`a -> { a.startsWith("test"); }`	Missing `return`
`a -> { return a.startsWith("test") }`	Missing semicolon

Introducing Functional Interfaces

- Lambdas work with **interfaces that have only one abstract method**. These are called **functional interfaces**.
- **@FunctionalInterface** : marks an interface as functional interface

```
public interface CheckTrait {
    boolean test(Animal a);
    }
```

Lambdas and Functional interface

```
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;

    public Animal(String speciesName, boolean
    hopper, boolean swimmer){
        species = speciesName;
        canHop = hopper;
        canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
    }
```

```
public interface CheckTrait {
    boolean test(Animal a);
    }
```

```
public class CheckIfHopper implements
CheckTrait {
    public boolean test(Animal a) {
        return a.canHop();
    }
    }
```

sourcemind

Lambdas and Functional interface

```java
import java.util.*;
public class TraditionalSearch {
    public static void main(String[] args) {
        // list of animals
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));
        // pass class that does check
        print(animals, new CheckIfHopper());
    }
    private static void print(List<Animal> animals, CheckTrait checker) {
        for (Animal animal : animals) {
        // the general check
        if (checker.test(animal))
            System.out.print(animal + " ");
        }
    System.out.println();
    }
    }
```

- Now what happens if we want to print the Animal s that swim? Sigh. We need to write another class, **CheckIfSwims** .

sourcemind

Lambdas and Functional interface

```
print(animals, new CheckIfHopper());
```

↓

```
print(animals, a -> a.canHop());
```

- How about Animals that cannot swim?

```
print(animals, a -> ! a.canSwim());
```

Built-in functional interfaces

Predicate , Consumer , Supplier , Function, Comparator, etc

PREDICATE

```
public interface Predicate<T> {
    boolean test(T t);
    }
```

CONSUMER

```
public interface Consumer<T> {
    void accept(T t);
    }
```

```
Consumer<String> consumer = x -> System.out.println(x);
```

sourcemind

SUPPLIER

```
public interface Supplier<T> {
    T get();
    }
```

```
Supplier<Integer> number = () -> 42;
Supplier<Integer> random = () ->new Random().nextInt();
```

COMPARATOR

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    }
```

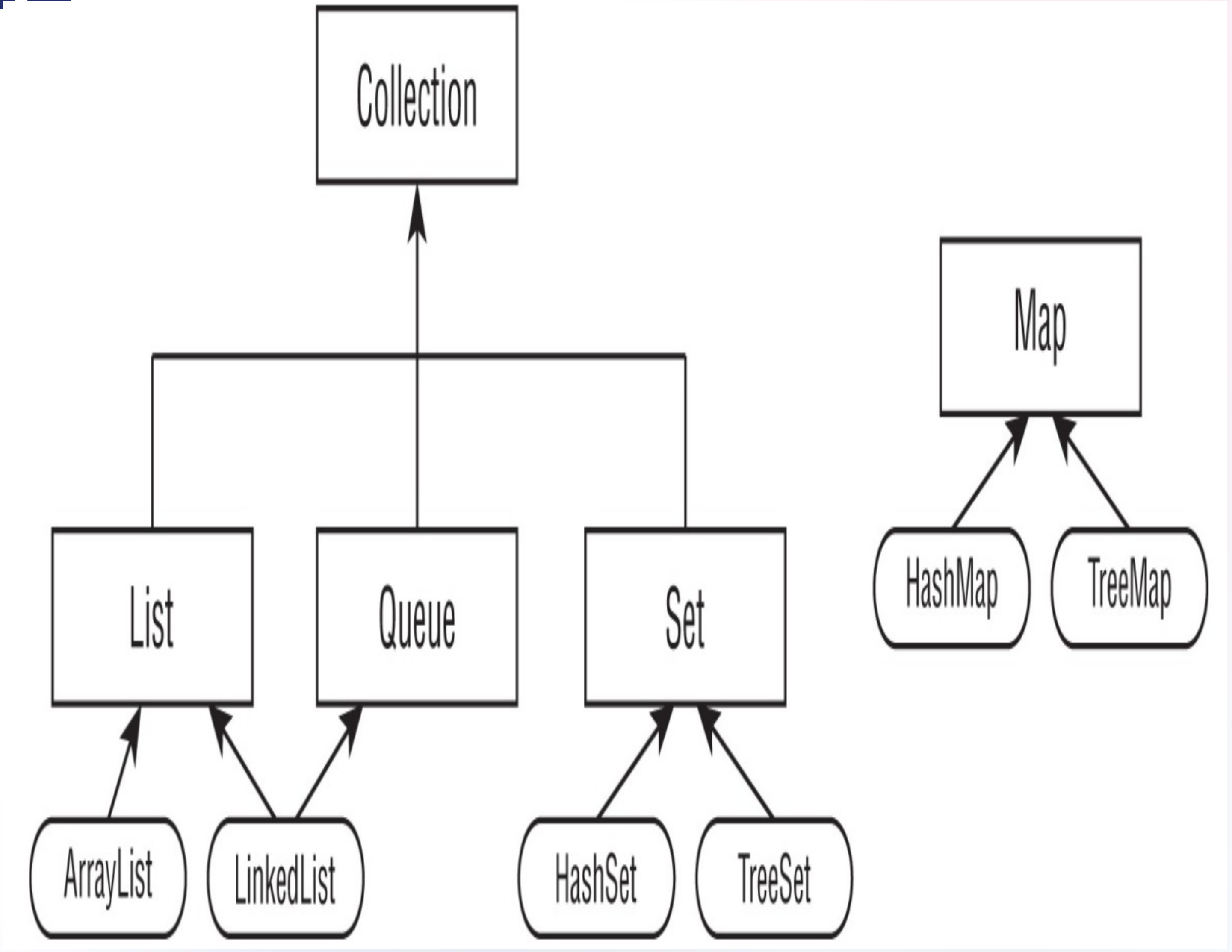
```
Comparator<Integer> ints = (i1, i2) -> i1 - i2;
```

FUNCTION

```
public interface Function<T, R> {
    R apply(T);
    }
```

Java Collections

- The **interfaces** are shown in **rectangles**, with the **classes** in **rounded boxes**
- The **Collection** interface is the root of all collections except maps



Four main interfaces

- **List** : A list is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an **int index**

- **Set** : A set is a collection that does **not allow duplicate** entries

- **Queue** : A queue is a collection that orders its elements in a **specific order for processing**. A typical queue processes its elements in a **first-in, first-out** order, but other orderings are possible

- **Map** : A map is a collection that maps keys to values, with **no duplicate keys** allowed. The elements in a map are **key/value pairs**

sourcemind

Examples

- Create a list

```
Collection<String> list1 = new ArrayList<>();

List<String> list2 = new ArrayList<>();

ArrayList<String> list3 = new ArrayList<>();
```

- Create a set

```
Collection<String> set1 = new HashSet<>();

Set<String> set2 = new HashSet<>();

HashSet<String> set3 = new HashSet<>();
```

COMMON COLLECTIONS METHODS

- The Collection interface contains useful methods for working with **lists**, **sets**, and **queues**

add()

- boolean add(E element)

```
ArrayList<String> list = new ArrayList<>();

System.out.println(list.add("Sparrow")); // true

System.out.println(list.add("Sparrow")); // true
```

remove()

- boolean remove(Object object)
- boolean remove(int index)

```
ArrayList<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk,hawk]

System.out.println(birds.remove("cardinal")); // false
System.out.println(birds.remove("hawk")); // true
System.out.println(birds); // [hawk]
```

isEmpty() and size()

- boolean isEmpty()
- int size()

```
ArrayList<String> birds = new ArrayList<>();
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
```

sourcemind

clear()

- void clear()

```
ArrayList<String> birds = new ArrayList<>();

birds.add("hawk"); // [hawk]

birds.add("hawk"); // [hawk, hawk]

System.out.println(birds.isEmpty()); // false

System.out.println(birds.size()); // 2

birds.clear(); // []

System.out.println(birds.isEmpty()); // true

System.out.println(birds.size()); // 0
```

contains()

- boolean contains(Object object)

```
ArrayList<String> birds = new ArrayList<>();

birds.add("hawk"); // [hawk]

System.out.println(birds.contains("hawk")); // true

System.out.println(birds.contains("robin")); // false
```

sourcemind

USING THE LIST INTERFACE

sourcemind

List Implementations

- **ArrayList** : an ArrayList is like a resizable array. When elements are added, the ArrayList automatically grows.

- **LinkedList** : it implements both List and Queue . It has all the methods of a List . It also has additional methods to facilitate adding or removing from the beginning and/or end of the list. LinkedList is a good choice when you'll be using it as Queue .

Factory methods to create a List

Method	Description	Can add elements?	Can replace element?	Can delete elements?
`Arrays.asList(varargs)`	Returns fixed size list backed by an array	No	Yes	No
`List.of(varargs)`	Returns immutable list	No	No	No
`List.copyOf(collection)`	Returns immutable list with copy of original collection's values	No	No	No

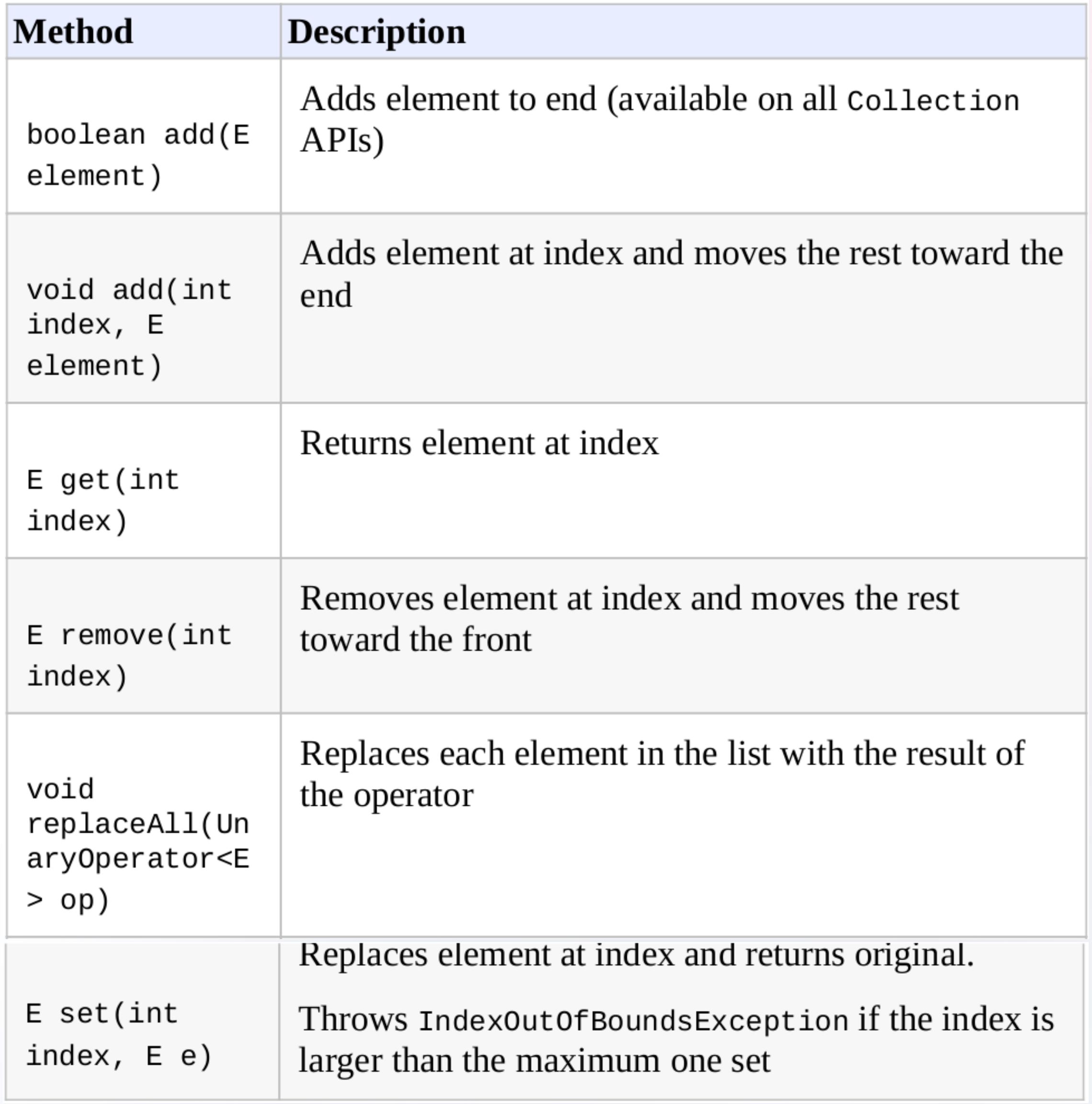
sourcemind

Factory methods to create a List

```java
String[] array = new String[] {"a", "b", "c"};

List<String> asList = Arrays.asList(array); // [a, b, c]

List<String> of = List.of(array); // [a, b, c]

List<String> copy = List.copyOf(asList); // [a, b, c]

array[0] = "z";

System.out.println(asList); // [z, b, c]

System.out.println(of); // [a, b, c]

System.out.println(copy); // [a, b, c]

asList.set(0, "x");

System.out.println(Arrays.toString(array)); // [x, b, c]

copy.add("y"); // throws UnsupportedOperationException
```

sourcemind

List methods



Method	Description
`boolean add(E element)`	Adds element to end (available on all `Collection` APIs)
`void add(int index, E element)`	Adds element at index and moves the rest toward the end
`E get(int index)`	Returns element at index
`E remove(int index)`	Removes element at index and moves the rest toward the front
`void replaceAll(UnaryOperator<E> op)`	Replaces each element in the list with the result of the operator
`E set(int index, E e)`	Replaces element at index and returns original. Throws `IndexOutOfBoundsException` if the index is larger than the maximum one set

ITERATING THROUGH A LIST

```
for (String string: list) {
    System.out.println(string);
    }
```

```
Iterator<String> iter = list.iterator();
while(iter.hasNext()) {
    String string = iter.next();
    System.out.println(string);
    }
```

USING THE SET INTERFACE

sourcemind

Set Implementations

- **HashSet** : a HashSet stores its elements in a hash table, which means the keys are a hash and the values are an Object . This means that it uses the hashCode() method of the objects to retrieve them more efficiently.

- **TreeSet** : a TreeSet stores its elements in a sorted tree structure.

Factory methods to create a Set

- Set<Character> letters = **Set.of**('z', 'o', 'o');
- Set<Character> copy = **Set.copyOf**(letters);

USING THE QUEUE INTERFACE

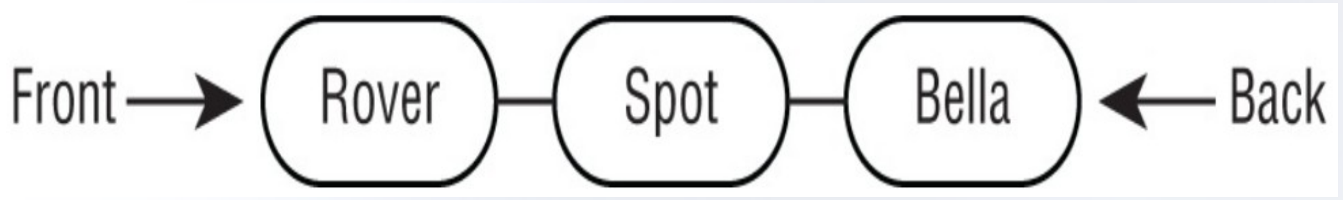
sourcemind

Queue Implementations

A queue is assumed to be **FIFO (first-in, first-out)**. Some queue implementations change this to use a different order. The other format is **LIFO (last-in, first-out)**, which is commonly referred to as a **stack**. In Java, though, both can be implemented with the **Queue** interface.

- **LinkedList** : a double-ended queue. you can insert and remove elements from both the front and back of the queue.
- **ArrayDeque** : more efficient double-ended queue.

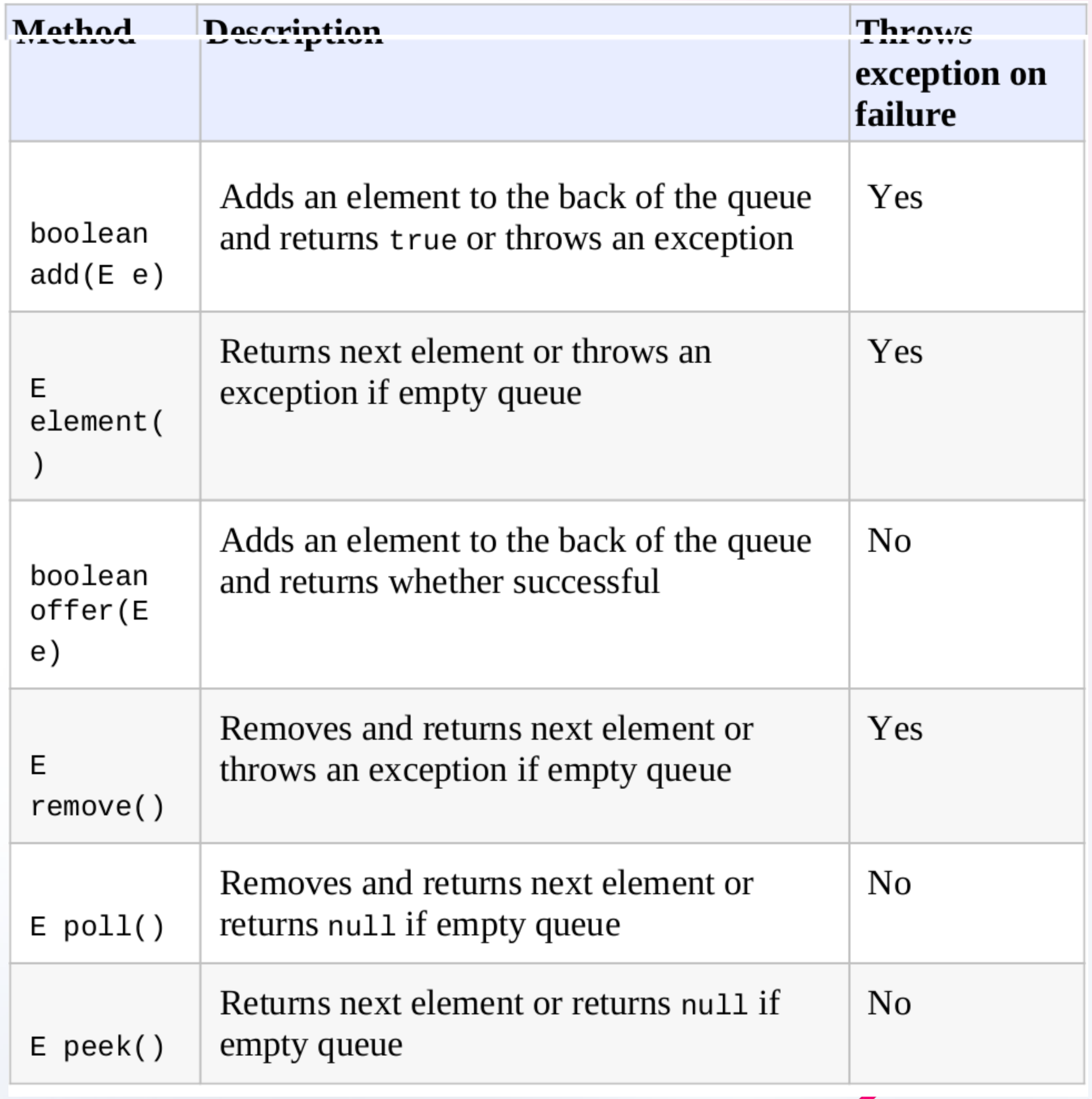
Example of a Queue :



sourcemind

Queue methods

The Queue interface contains many methods. Theses are some.

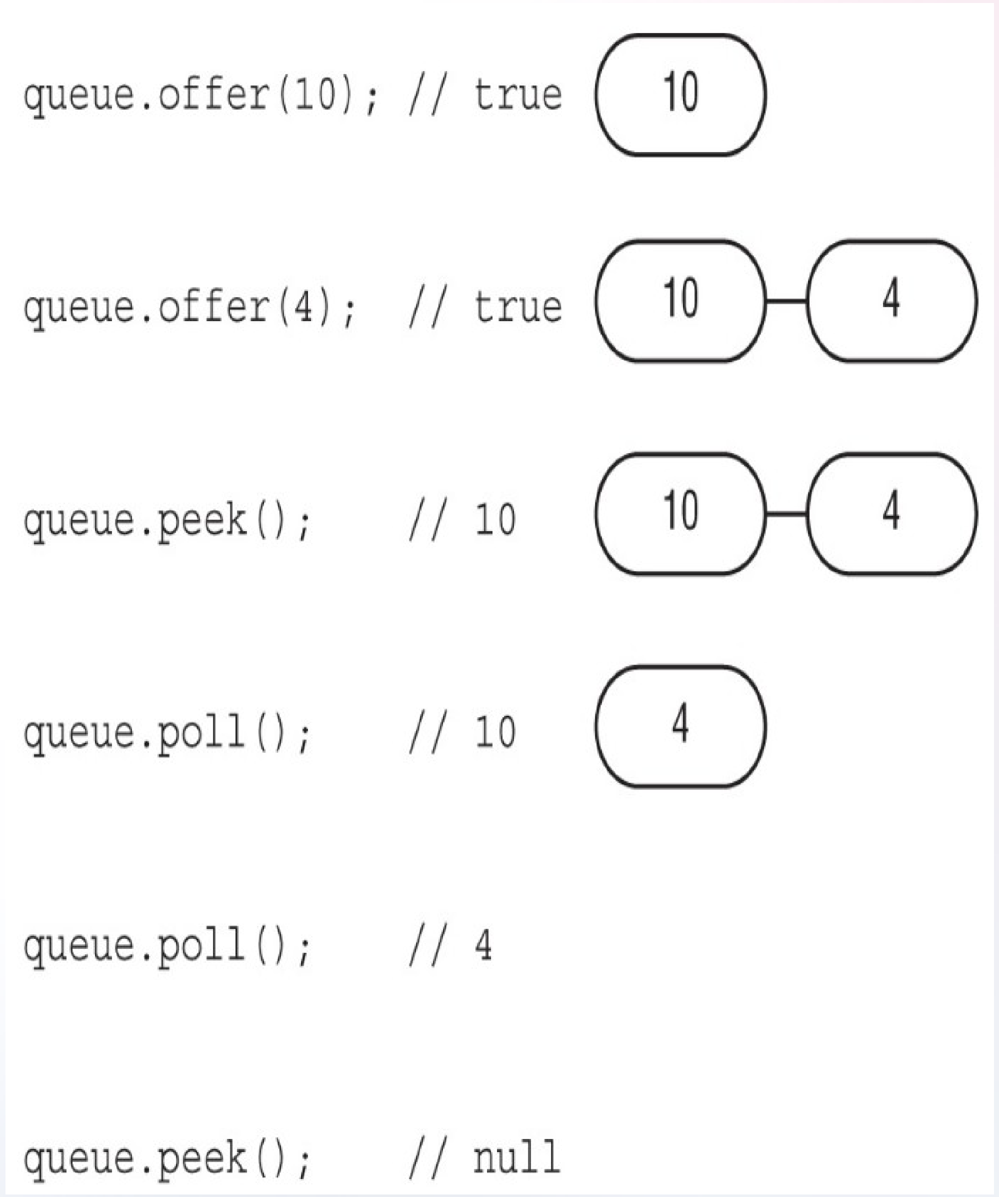


Method	Description	Throws exception on failure
`boolean add(E e)`	Adds an element to the back of the queue and returns `true` or throws an exception	Yes
`E element()`	Returns next element or throws an exception if empty queue	Yes
`boolean offer(E e)`	Adds an element to the back of the queue and returns whether successful	No
`E remove()`	Removes and returns next element or throws an exception if empty queue	Yes
`E poll()`	Removes and returns next element or returns `null` if empty queue	No
`E peek()`	Returns next element or returns `null` if empty queue	No

sourcemind

Example

```
Queue<Integer> queue = new LinkedList<>();

System.out.println(queue.offer(10)); // true

System.out.println(queue.offer(4)); // true

System.out.println(queue.peek()); // 10

System.out.println(queue.poll()); // 10

System.out.println(queue.poll()); // 4

System.out.println(queue.peek()); // null
```


```
queue.offer(10); // true
queue.offer(4);  // true
queue.peek();    // 10
queue.poll();    // 10
queue.poll();    // 4
queue.peek();    // null
```

USING THE MAP INTERFACE

sourcemind

Map Implementations

- **HashMap** : a HashMap stores the keys in a hash table. This means that it uses the hashCode() method of the keys to retrieve their values more efficiently.
- **LinkedHashMap** : a kind of HashMap that maintains keys insertion order.
- **TreeMap** : A TreeMap stores the keys in a sorted tree structure.

Factory methods to create a Map

- MAP.OF() and MAP.COPYOF()

```
    Map.of("key1", "value1", "key2", "value2");
    Map.of("key1", "value1", "key2"); // INCORRECT
```

- Map.ofEntries(
-     Map.entry("key1", "value1"),Map.entry("key1", "value1"))

sourcemind

Map methods

Method	Description
`void clear()`	Removes all keys and values from the map.
`boolean containsKey(Object key)`	Returns whether key is in map.
`V remove(Object key)`	Removes and returns value mapped to key. Returns `null` if none.
`V replace(K key, V value)`	Replaces the value for a given key if the key is set. Returns the original value or `null` if none.
`void replaceAll(BiFunction<K, V, V> func)`	Replaces each value with the results of the function.
`int size()`	Returns the number of entries (key/value pairs) in the map.
`Collection<V> values()`	Returns `Collection` of all values.
`boolean containsValue(Object value)`	Returns whether value is in map.
`Set<Map.Entry<K,V>> entrySet()`	Returns a `Set` of key/value pairs.
`void forEach(BiConsumer(K key, V value))`	Loop through each key/value pair.
`V get(Object key)`	Returns the value mapped by key or `null` if none is mapped.
`V getOrDefault(Object key, V defaultValue)`	Returns the value mapped by the key or the default value if none is mapped.
`boolean isEmpty()`	Returns whether the map is empty.
`Set<K> keySet()`	Returns set of all keys.
`V merge(K key, V value, Function(<V, V, V> func))`	Sets value if key not set. Runs the function if the key is set to determine the new value. Removes if `null`.
`V put(K key, V value)`	Adds or replaces key/value pair. Returns previous value or `null`.
`V putIfAbsent(K key, V value)`	Adds value if key not present and returns null. Otherwise, returns existing value.

sourcemind

Example

```java
Map<String, String> map = new HashMap<>();
map.put("koala", "bamboo");
map.put("lion", "meat");
map.put("giraffe", "leaf");
String food = map.get("koala"); // bamboo
for (String key: map.keySet())
        System.out.print(key + ","); // koala,giraffe,lion,
```

OLDER COLLECTIONS

- **Vector** : Implements List . If you don't need concurrency, use **ArrayList** instead.
- **Hashtable** : Implements Map . If you don't need concurrency, use **HashMap** instead.
- **Stack** : Implements Queue . If you don't need concurrency, use a **LinkedList** instead.

There are modern alternatives implementations to deal with concurrency.

Exercise

- Collect unknown number of items from input and sort them using a List
- Collect unknown number of integer values from input, sort them, and eliminate the duplicate values
- Collect some String input values from user and group them by the starting letter

Generics

Generics in Java enable “types” to be parameters in classes, interfaces, and methods.

GENERIC CLASSES

- You can introduce generics into your own classes. The syntax for introducing a generic is to declare a formal type parameter in angle brackets.

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }
    public void packCrate(T contents) {
        this.contents = contents;
    }
}
```

The generic type **T** is available anywhere within the Crate class. When you instantiate the class, you tell the compiler what T should be for that particular instance.

NAMING CONVENTIONS FOR GENERICS

- **E** for an element
- **K** for a map key
- **V** for a map value
- **N** for a number
- **T** for a generic data type
- **S , U , V** , and so forth for multiple generic types

sourcemind

Example

```
Elephant elephant = new Elephant();

Crate<Elephant> crateForElephant = new Crate<>();

crateForElephant.packCrate(elephant);

Elephant inNewHome =
crateForElephant.emptyCrate();
```

```
Crate<Zebra> crateForZebra = new Crate<>();
```

sourcemind

Two generic parameters example

```
public class SizeLimitedCrate<T, U> {
    private T contents;
    private U sizeLimit;
    public SizeLimitedCrate(T contents, U sizeLimit) {
        this.contents = contents;
        this.sizeLimit = sizeLimit;
    }
    }
```

```
Elephant elephant = new Elephant();
Integer numPounds = 15_000;
SizeLimitedCrate<Elephant, Integer> c1 = new SizeLimiteCrate<>(elephant, numPounds);
```

sourcemind

TYPE ERASURE

Behind the scenes, the compiler replaces generics type with Object .

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }
    public void packCrate(T contents) {
        this.contents = contents;
    }
}
```

```
public class Crate {
    private Object contents;
    public Object emptyCrate() {
        return contents;
    }
    public void packCrate(Object contents) {
        this.contents = contents;
    }
}
```

sourcemind

TYPE ERASURE

- This means there is only one class file. There aren't different copies for different parameterized types. (Some other languages work that way.)
- This process of removing the generics syntax from your code is referred to as **type erasure**. Type erasure allows your code to be compatible with older versions of Java that do not contain generics.
- The compiler adds the relevant casts for your code to work with this type of erased class. For example, you type the following:

- **Robot r = crate.emptyCrate();**
- The compiler turns it into the following:
- **Robot r = (Robot) crate.emptyCrate();**

sourcemind

GENERIC INTERFACES

```
public interface Shippable<T> {
    void ship(T t);
    }
```

```
class ShippableRobotCrate implements Shippable<Robot> {
    public void ship(Robot t) { }
    }
```

```
class ShippableAbstractCrate<U> implements Shippable<U> {
    public void ship(U t) { }
    }
```

```
class ShippableCrate implements Shippable {
    public void ship(Object t) { }
    }
```

sourcemind

WHAT YOU CAN'T DO WITH GENERIC TYPES

- **Calling a constructor**: Writing new T() is not allowed because at runtime it would be new Object() .
- **Creating an array of that generic type**: This one is the most annoying, but it makes sense because you'd be creating an array of Object values.
- **Calling instanceof** : This is not allowed because at runtime List<Integer> and List<String> look the same to Java thanks to type erasure.
- **Using a primitive type as a generic type parameter**: This isn't a big deal because you can use the wrapper class instead. If you want a type of int , just use Integer .
- **Creating a static variable as a generic type parameter**: This is not allowed because the type is linked to the instance of the class.

sourcemind

GENERIC METHODS

```
public class Handler {
    public static <T> void prepare(T t) {
        System.out.println("Preparing " + t);
    }
    public static <T> Crate<T> ship(T t) {
        System.out.println("Shipping " + t);
        return new Crate<T>();
    }
    }
```

sourcemind

OPTIONAL SYNTAX FOR INVOKING A GENERIC METHOD

You can call a generic method normally, and the compiler will try to figure out which one you want. Alternatively, you can specify the type explicitly to make it obvious what the type is.

```
Box.<String>ship("package");

Box.<String[]>ship(args);
```

sourcemind

Exercise

Collect unknown number of items from input and sort them using a List

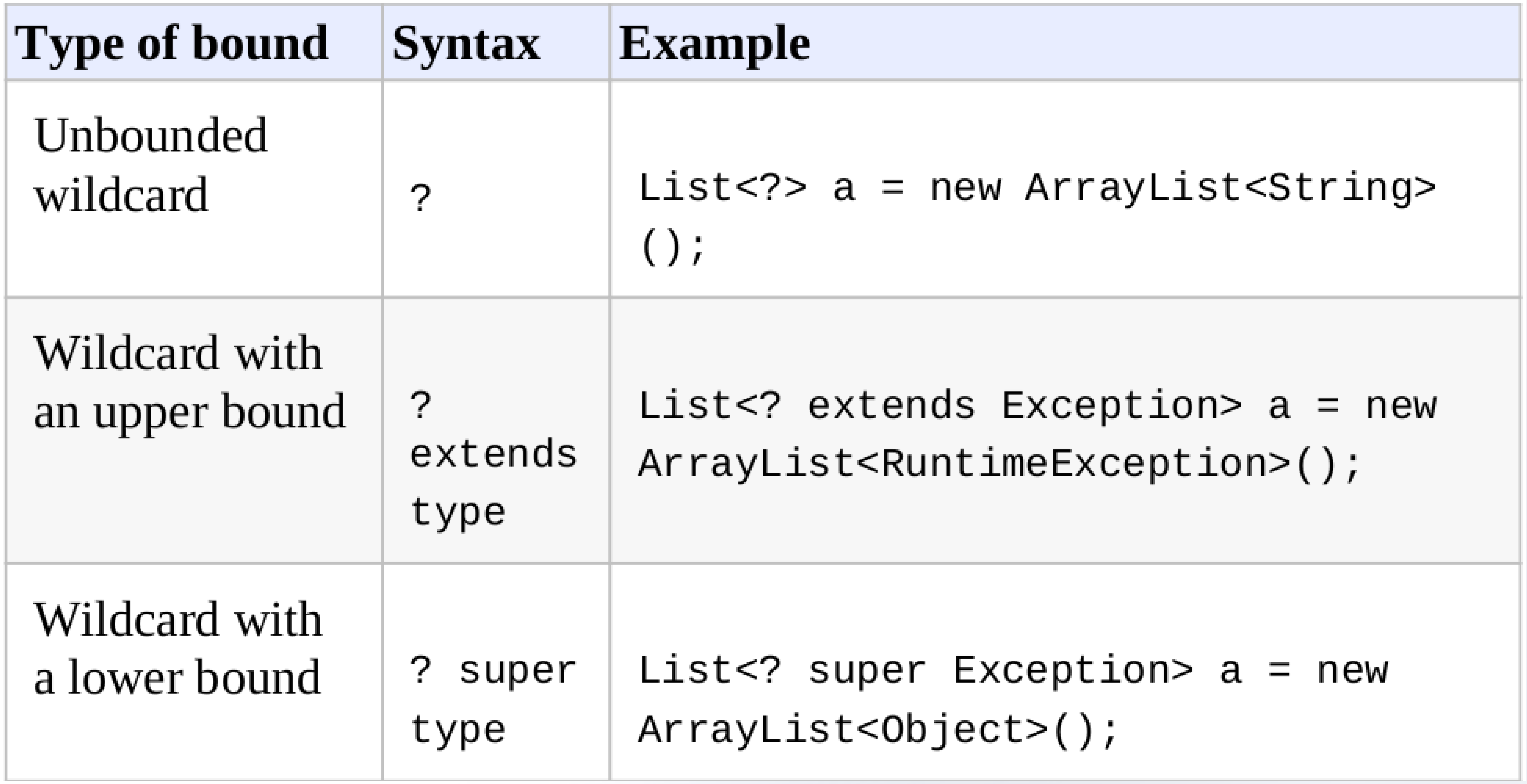
Collect unknown number of integer values from input, sort them, and eliminate the duplicatevalues

Collect some String input values from user and group them by the starting letter

BOUNDING GENERIC TYPES

- A **bounded parameter type** is a generic type that specifies a bound for the generic.
- A **wildcard generic type** is an unknown generic type represented with a question mark (**?**).

Types of bounds



Type of bound	Syntax	Example
Unbounded wildcard	`?`	`List<?> a = new ArrayList<String>();`
Wildcard with an upper bound	`? extends type`	`List<? extends Exception> a = new ArrayList<RuntimeException>();`
Wildcard with a lower bound	`? super type`	`List<? super Exception> a = new ArrayList<Object>();`

Unbounded Wildcards

```
public static void printList(List<?> list) {
    for (Object x: list)
    System.out.println(x);
    }
    public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords);
    }
```

sourcemind

Upper-Bounded Wildcards

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
```

```
List<? extends Number> list = new ArrayList<Integer>();
```

The **upper-bounded** wildcard says that any class that **extends Number or Number** itself can be used as the formal parameter type.

Lower-Bounded Wildcards

```
public static void addSound(List<? super String> list) {
    list.add("quack");
    }
```

```
List<String> strings = new ArrayList<String>();
strings.add("tweet");
List<Object> objects = new ArrayList<Object>(strings);
addSound(strings);
addSound(objects);
```

With a lower bound, we are telling Java that the list will be a list of String objects or a list of some objects that are a superclass of String .

sourcemind

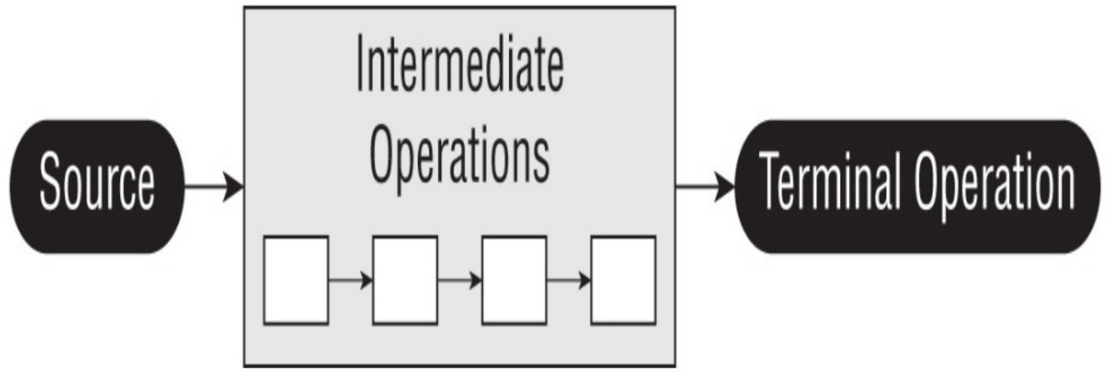
Streams

sourcemind

Streams

- A **stream** in Java is a sequence of data.
- A **stream pipeline** consists of the operations that run on a stream to produce a result

Stream pipeline



- **Source**: Where the stream comes from
- **Intermediate operations**: Transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
- **Terminal operation**: Actually produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.

CREATING STREAM SOURCES

In Java, the streams we have been talking about are represented by the **Stream<T>** interface, defined in the **java.util.stream** package.

Creating Finite Streams

```
Stream<String> empty = Stream.empty(); // count = 0

Stream<Integer> singleElement = Stream.of(1); // count = 1

Stream<Integer> fromArray = Stream.of(1, 2, 3); // count = 3
```

Java also provides a convenient way of converting a Collection to a stream.

```
var list = List.of("a", "b", "c");

Stream<String> fromList = list.stream();
```

sourcemind

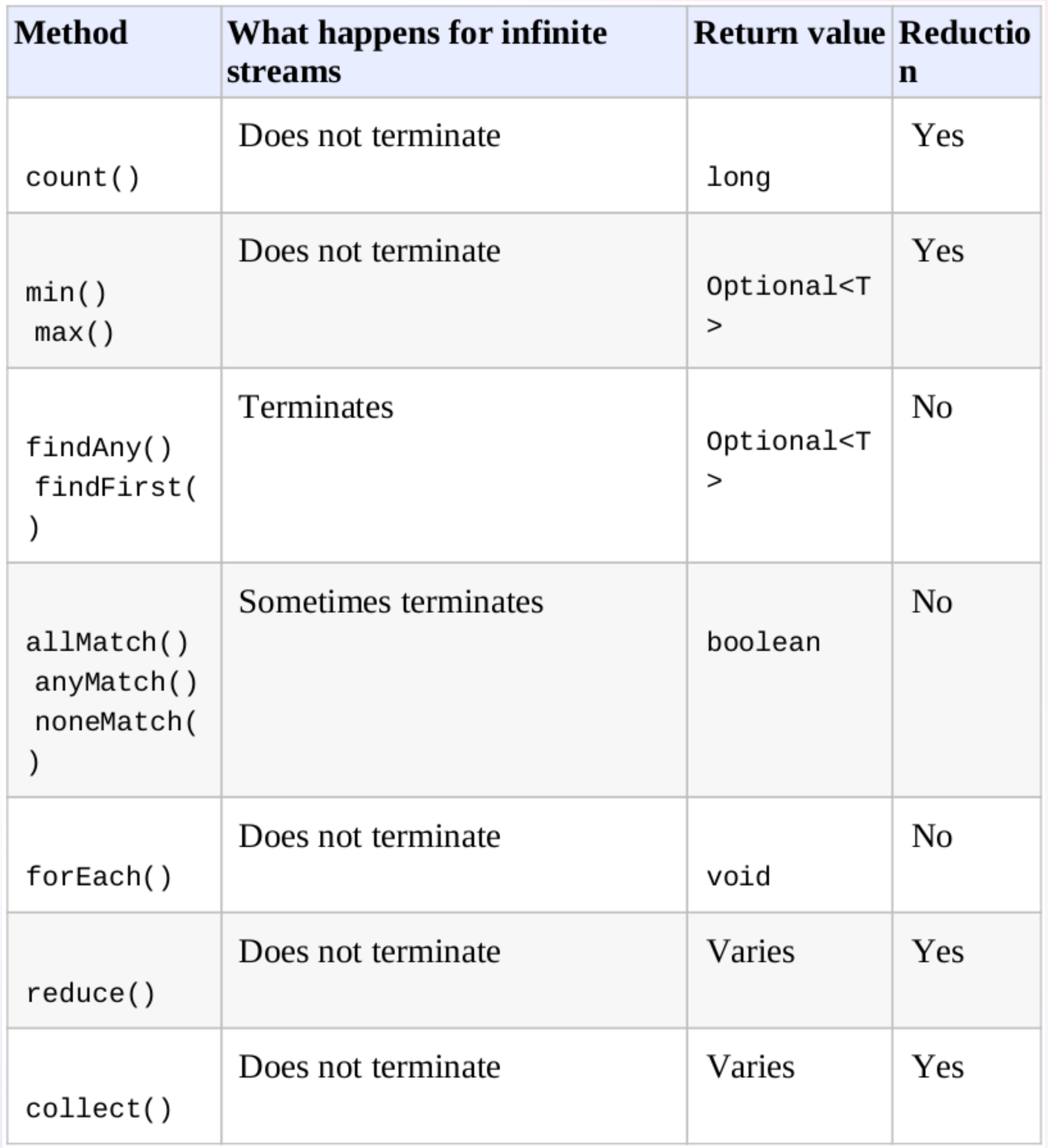
Creating Infinite Streams

```
Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

sourcemind

TERMINAL OPERATIONS

- You can perform a terminal operation without any intermediate operations but not the other way around.
- **Reductions** are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or Object .



Method	What happens for infinite streams	Return value	Reduction
`count()`	Does not terminate	`long`	Yes
`min()` `max()`	Does not terminate	`Optional<T>`	Yes
`findAny()` `findFirst()`	Terminates	`Optional<T>`	No
`allMatch()` `anyMatch()` `noneMatch()`	Sometimes terminates	`boolean`	No
`forEach()`	Does not terminate	`void`	No
`reduce()`	Does not terminate	Varies	Yes
`collect()`	Does not terminate	Varies	Yes

sourcemind

Examples : count(), min(), max()

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
System.out.println(s.count()); // 3
```

```
Optional<T> min(Comparator<? super T> comparator)
Optional<T> max(Comparator<? super T> comparator)
```

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());
min.ifPresent(System.out::println); // ape
```

Examples : reduce ()

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

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

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

```
Stream<String> stream = Stream.of("w", "o", "l", "f");

String word = stream.reduce("", (s, c) -> s + c);

System.out.println(word); // wolf
```

sourcemind

Examples : collect()

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R>  combiner)
<R,A> R collect(Collector<? super T, A,R> collector)
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
StringBuilder word = stream.collect(
StringBuilder::new,
StringBuilder::append,
StringBuilder::append)
System.out.println(word); // wolf
```

sourcemind

Examples : collect()

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set = stream.collect(
TreeSet::new,
TreeSet::add,
TreeSet::addAll);
System.out.println(set); // [f, l, o, w]
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set =
stream.collect(Collectors.toCollection(TreeSet::new));System.out.println(set); // [f, l, o, w]
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(Collectors.toSet());
System.out.println(set); // [f, w, l, o]
```

sourcemind

INTERMEDIATE OPERATIONS

- Unlike a terminal operation, an intermediate operation produces a stream as its result. An intermediate operation can also deal with an infinite stream simply by returning another infinite stream.

filter(), distinct(), limit(), skip(), map(), flatMap(), sorted(), peek()

Examples : filter()

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.filter(x -> x.startsWith("m"))
.forEach(System.out::print); // monkey
```

Examples : distinct()

```
Stream<T> distinct()
```

```
Stream<String> s = Stream.of("duck", "duck", "duck", "goose");
s.distinct()
.forEach(System.out::print); // duckgoose
```

sourcemind

Examples : limit() and skip()

```
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(5)
.limit(2)
.forEach(System.out::print); // 67
```

sourcemind

Examples : map()

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length)
.forEach(System.out::print); // 676
```

sourcemind

Examples : peek()

```
Stream<T> peek(Consumer<? super T> action)
```

```
var stream = Stream.of("black bear", "brown bear", "grizzly");
long count = stream.filter(s -> s.startsWith("g"))
.peek(System.out::println).count(); // grizzly
System.out.println(count); // 1
```

sourcemind

PRIMITIVE STREAMS

IntStream : Used for the primitive types **int , short , byte , and char**

LongStream : Used for the primitive type **long**

DoubleStream : Used for the primitive types **double and float**

```
IntStream intStream = IntStream.of(1, 2, 3);
OptionalDouble avg = intStream.average();
System.out.println(avg.getAsDouble()); // 2.0
```

Primitive stream methods

Method	Primitive stream	Description
OptionalDouble average()	IntStream, LongStream, DoubleStream	The arithmetic mean of the elements
Stream<T> boxed()	IntStream, LongStream, DoubleStream	A Stream<T> where T is the wrapper class associated with the primitive value
OptionalInt max(), min()	IntStream	The maximum element of the stream or The minimum element of the stream
OptionalLong max(), min()	LongStream	
OptionalDouble max(), min()	DoubleStream	

sourcemind

Primitive stream methods

IntStream range(int a, int b)	IntStream	Returns a primitive stream from a (inclusive) to b (exclusive)
LongStream range(long a, long b)	LongStream	
IntStream rangeClosed(int a, int b)	IntStream	Returns a primitive stream from a (inclusive) to b (inclusive)
LongStream rangeClosed(long a, long b)	LongStream	

sourcemind

Primitive stream methods

int sum()	IntStreamIntStream	Returns the sum of the elements in the stream
long sum()	LongStream	
double sum()	DoubleStream	
IntSummaryStatistics summaryStatistics()	IntStream	Returns an object containing numerous stream statistics such as the average, min, max, etc.
LongSummaryStatistics summaryStatistics()	LongStream	
DoubleSummaryStatistics summaryStatistics()	DoubleStream	

Examples

```
DoubleStream oneValue = DoubleStream.of(3.14);
oneValue.forEach(System.out::println);

DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
varargs.forEach(System.out::println);
```

```
var random = DoubleStream.generate(Math::random);
var fractions = DoubleStream.iterate(.5, d -> d / 2);
random.limit(3).forEach(System.out::println);
fractions.limit(3).forEach(System.out::println);
```

sourcemind

End

sourcemind