IT資格道場

Java Gold SE 25(1Z0-828-JPN) 学習用テキスト

Silver の先へ —— 読んで理解する1冊目

無料サンプル(第1章まで)

Oracle公式 Java SE 25 Programmer II(1Z0-828-JPN) 試験内容チェックリスト 準拠
2026年11月版

SAMPLE

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

SAMPLE

目次

目次

この試験が問うもの

第1章 コレクションとジェネリクス [1.1/1.2/1.3/1.4/1.5/1.6]

- オート・ボクシングの使用 [1.1]
- コレクション・フレームワークの概要と主要なインタフェース [1.2]
- Sequenced Collectionsの使用 [1.3]
- 総称型の使用 [1.4]
- Comparatorインタフェースを使用したコレクションのソート [1.5]
- コレクション用の便利なメソッドの使用 [1.6]

サンプルはここまでです

SAMPLE

3 冊の使い方は、ダウンロード欄の案内を参照してください。

Java SE 25 Programmer II（1Z0-828-JPN）の全 8 主題・45 細目を、主題ごとに 1 章ずつ、出題範囲の順に並べた学習用テキストです。各章は、細目ごとの節と、その中の小見出しでできています。載せているコードの出力・例外・コンパイルエラーは、JDK 25 の `javac --release 25` と `java` で実際に動かして確かめました。API の仕様は、Java SE 25 の公式文書の脚注で確かめられます。脚注は巻末の「商標と出典」にまとめています。

SAMPLE

目次

- 第1章 コレクションとジェネリクス [1.1/1.2/1.3/1.4/1.5/1.6]
- 第2章 関数型インタフェースとラムダ式 [2.1/2.2/2.3/2.4]
- 第3章 JavaストリームAPI [3.1/3.2/3.3/3.4/3.5/3.6/3.7/3.8/3.9/3.10]
- 第4章 Javaモジュール・システム [4.1/4.2/4.3/4.4/4.5/4.6/4.7]
- 第5章 並列処理 [5.1/5.2/5.3/5.4/5.5/5.6/5.7]
- 第6章 ファイル I/O [6.1/6.2/6.3/6.4/6.5]
- 第7章 JDBC [7.1/7.2/7.3]
- 第8章 ローカライズ [8.1/8.2/8.3]
- 巻末：商標と出典

SAMPLE

この試験が問うもの

項目	内容
試験名	Java SE 25 Programmer II（1Z0-828-JPN）
問題数	60 問（選択問題）
試験時間	120 分
合格ライン	65%
出題範囲	8 セクション・45 細目
前提資格	Oracle Certified Java Programmer, Silver SE 25（または Java SE 7 以降の Silver）

設問は、コードやコマンドを見せて結果を選ばせる形が中心です。コンパイルが通るか、何が出力されるか、どの例外が出るかを、手を動かさずに読み切る力が問われます。

選択肢の「〜で異常終了する」は、`main` の中で捕まえなかった例外が、スタックトレースを標準エラー出力に出して、プログラムを異常終了させる動きを指します。`main` 以外のスレッドで捕まえなかった例外は、そのスレッドだけを終わらせます。プログラム自体は止まりません。

SAMPLE

第1章 コレクションとジェネリクス [1.1/1.2/1.3/1.4/1.5/1.6]

- オート・ボクシングの使用 [1.1]
- コレクション・フレームワークの概要と主要なインタフェース [1.2]
- Sequenced Collectionsの使用 [1.3]
- 総称型の使用 [1.4]
- Comparatorインタフェースを使用したコレクションのソート [1.5]
- コレクション用の便利なメソッドの使用 [1.6]

オート・ボクシングの使用 [1.1]

ボクシングは、基本データ型の値を対応する参照型のオブジェクトとして扱う変換です。逆向きの変換がアンボクシングで、どちらもコンパイラが自動で挿入します。[1] 対応は次の8組です。

boolean	byte	short	char	int	long	float	double
Boolean	Byte	Short	Character	Integer	Long	Float	Double

SAMPLE

```
Integer a = 5;            // ボクシング
int b = a;                // アンボクシング
Integer c = a + 1;        // a を取り出して加算し、結果を包み直す
Integer s = 10; s++; s += 5;   // s は 16
Long l = 1;               // コンパイルエラー（int から Long へは変換できない）
Long ok = 1L;
```

ボクシングは基本データ型から対応する1つの参照型への変換です。int の値は Integer になれても、Long や Double には包まれません。代入でも同じで、`Long l = 1;` は通りません。

自動変換が起きる場所と equals

自動変換が働く代表的な場所は、代入、メソッド呼び出しの引数、演算の3つです。`list.add(5)` は 5 を Integer に包んで渡します。`a + 1` のような演算は Integer をいったん int に戻して計算します。

`equals` は型まで比べます。`Integer.equals` は、引数が null でない Integer で、同じ int の値を持つ時だけ true を返します（それ以外は false）。[2]

```
Long L = 127L;
L.equals(127)    // false（引数は Integer に包まれる）
L.equals(127L)   // true
Double D = 3.0;
D.equals(3)      // false（Integer が渡る）
```

SAMPLE

`Set.contains` と `Map.get` の引数の型は `Object` です。int を渡すと Integer に包まれて渡るので、`Long` の要素やキーとは一致しません。`Set<Long> s` に 5L が入っていても `s.contains(5)` は false、`Map<Long, String>` の `get(5)` は null です（`contains(5L)` なら true）。[3]

null のアンボクシングと == の比較

参照が null のままアンボクシングすると NullPointerException が出ます（null は基本データ型の値に直せない）。[4] 次の例では、前の4つが実行時に NullPointerException になり、最後の1つは null のまま通ります。

```
Integer x = null;
int y = x;                          // NullPointerException
if (x == 0) { }                     // 比較のために取り出して NullPointerException
int v = new HashMap<String,Integer>().get("no");   // get は null を返し、int への代入で NullPointerException
Integer z = true ? x : 0;           // 一方が int なので x を取り出し、NullPointerException
Integer w = true ? x : Integer.valueOf(0);          // 両方とも Integer なので取り出さず、w は null
```

`==` は2つの参照型どうしなら同じオブジェクトかどうかを比べ、片方が基本データ型なら値を比べます。Integer の `valueOf(int)` は -128 から 127 の範囲を必ずキャッシュして同じ参照を返します。ボクシングで得る Integer も同じ規則に従います。[5] この範囲の外は、キャッシュするかどうかを API が約束していません。JDK 25 で動かすと、範囲の外は別の参照になります（128 は false）。

SAMPLE

```
Integer i1 = 127, i2 = 127, j1 = 128, j2 = 128;
i1 == i2        // true
j1 == j2        // false
j1.equals(j2)   // true
j1 == 128       // true（片方が int なので値を比べる）
```

値を比べる時は equals を使います。参照を比べる == に頼ると、127 までは動いて 128 から動かなくなるコードになります。

オーバーロードの解決とボクシング

メソッド呼び出しの候補選びは3段階で進みます（1段で見つかれば、次の段へは進まない）。[6]

1. 第1段: ボクシング・アンボクシングと可変長引数を使わずに探す
2. 第2段: ボクシングとアンボクシングを許して探す（可変長引数はまだ使わない）
3. 第3段: 可変長引数も候補に入れて探す

拡大変換（int から long など）は第1段で、ボクシングは第2段で、可変長引数は第3段で現れます。

呼び出し	候補	選ばれる方	理由
`m(5)`	`m(long)` と `m(Integer)`	`m(long)`	拡大変換が第1段
`n(5)`	`n(Object)` と `n(int...)`	`n(Object)`	ボクシングが第2段、可変長引数は第3段

SAMPLE

呼び出し	候補	選ばれる方	理由
`p(5)`	`p(Integer)` と `p(int...)`	`p(Integer)`	同上
`q(5)`	`q(Long)` と `q(Object)`	`q(Object)`	int は Long に包めず、Object なら包める
`r(5)`	`r(double)` と `r(Object)`	`r(double)`	拡大変換が第1段
`m(i)`（i は Integer の変数）	`m(long)` と `m(Integer)`	`m(Integer)`	そのままの型が第1段。アンボクシングして long に広げるのは第2段
`r(i)`（i は Integer の変数）	`r(double)` と `r(Object)`	`r(Object)`	参照の拡大（Integer から Object）が第1段。アンボクシングは第2段

上の表は、7つの呼び出しの結果です。`q(5)` は「5 を long に広げてから Long に包む」とは進みません。拡大変換の後にボクシングを続ける変換は無く、ボクシングの後に参照の拡大変換（Integer から Object）を続ける変換はあります。`q(Object)` が選ばれるのは、この後の形です。

間違えやすい組み合わせが `List<Integer>` の `remove` です。

SAMPLE

```
List<Integer> li = new ArrayList<>(List.of(10, 20, 30, 1));
li.remove(1);                    // remove(int): 位置1を削除 → [10, 30, 1]
li.remove(Integer.valueOf(1));   // remove(Object): 値1を削除 → [10, 30]
li.remove((Object) 30);          // 値30を削除 → [10]
```

引数が int なら第1段で `remove(int)` が選ばれ、ボクシングは起きません。値で消したい時は Integer に包んで渡します（`Integer.valueOf(1)` か `(Object) 30`）。

コレクション・フレームワークの概要と主要なインタフェース [1.2]

コレクション・フレームワークの中心は Collection インタフェースです。その下に、順序と重複を扱う List、重複を持たない Set、先頭から取り出す Queue が並び、Queue の下に両端を操作する Deque があります。Map は Collection の子ではなく別の系統で、キーと値の対応を持ちます。

インタフェース	重複	順序	主な実装
List	許す	入れた位置で決まる（添字あり）	ArrayList、LinkedList
Set	許さない	実装による	HashSet、LinkedHashSet、TreeSet
Queue / Deque	許す	先頭と末尾	ArrayDeque、LinkedList、PriorityQueue

SAMPLE

インタフェース	重複	順序	主な実装
Map	キーは許さない	実装による	HashMap、LinkedHashMap、TreeMap

Collection と List の操作

List は順序を持つコレクションで、位置（添字）で要素を操作でき、重複を許します。[7] 範囲外の添字は IndexOutOfBoundsException になります（5要素のリストの `get(10)` は「Index 10 out of bounds for length 5」）。

```
List<Integer> l = new ArrayList<>(List.of(5, 6, 7));
l.add(8)         // true（末尾に追加）
l.indexOf(7)     // 2
l.set(0, 50)     // 50 に置き換え、前の値 5 を返す
l.add(1, 99);    // 位置1に挿入 → [50, 99, 6, 7, 8]
l.subList(1, 3)  // [99, 6]（from を含み to を含まない）
```

`remove(int)` は指定した位置の要素を取り除き、`remove(Object)` は最初に見つかった等しい要素を取り除きます。`List<Integer>` ではどちらが選ばれるかが引数の型で決まります（int なら位置、Integer なら値）。

作り方で変更の可否が変わります。

作り方	add / remove	set	null の要素
`new ArrayList<>(...)`	できる	できる	入れられる
`List.of(...)`	UnsupportedOperationException	UnsupportedOperationException	NullPointerException

SAMPLE

作り方	add / remove	set	null の要素
`Arrays.asList(...)`	UnsupportedOperationException	できる（元の配列も変わる）	入れられる

`List.of` が返す不変のリストは、追加・削除・置き換えのどれもできません。変更のメソッドは UnsupportedOperationException を投げます。[8] null の要素も許しません（`List.of(1, null)` は NullPointerException）。`List.of(1).add(null)` は、渡す値が null でも UnsupportedOperationException です。[8] `removeIf` は条件に合う要素が無くても UnsupportedOperationException です。

`Arrays.asList` は、元の配列に支えられた固定長のリストを返します。[9] set はできます（元の配列にも書き戻される）。大きさを変えるメソッド（`add`・`remove`・要素を取り除く `removeIf` など）は、リストを変えずに UnsupportedOperationException を投げます。[9]

```
String[] arr = {"x", "y", "z"};
List<String> al = Arrays.asList(arr);
al.set(0, "A");        // arr も {"A","y","z"} になる
arr[1] = "B";          // al も [A, B, z] になる
al.add("w");           // UnsupportedOperationException
al.removeIf(s -> s.equals("A"));     // UnsupportedOperationException
Arrays.asList(new int[]{1,2,3}).size()  // 1（int[] 全体が1要素）
Arrays.asList(1, 2, 3).size()           // 3
```

int の配列を渡すと、配列そのものが1つの要素になります。要素を3つ持つリストにしたい時は Integer の配列か、個別の値で渡します。

SAMPLE

Set の実装の違い

Set は `e1.equals(e2)` となる組を持たないコレクションです。[10]

実装	走査の順序	null	重複の判定
HashSet	保証しない	1つだけ入る	hashCode と equals
LinkedHashSet	入れた順（挿入順）	1つだけ入る	hashCode と equals
TreeSet	並び順（自然順序か Comparator）	NullPointerException	compareTo か compare が 0

HashSet は走査の順序を保証しません。LinkedHashSet は入れた順で走査し、すでにある要素をもう一度入れても位置は変わりません。[11]

```
Set<String> ls = new LinkedHashSet<>();
ls.add("banana"); ls.add("apple"); ls.add("banana"); ls.add("cherry");
ls            // [banana, apple, cherry]
ls.add("apple")   // false（すでにある）
new TreeSet<>(ls) // [apple, banana, cherry]
Set<String> tc = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
tc.add("Apple"); tc.add("APPLE")  // 2回目は false、tc は [Apple]
```

TreeSet は equals ではなく比較の結果が 0 かどうかで重複を決めます。比較が 0 を返す2つの要素は、equals が false でも後から入れた方が捨てられます。

HashSet と HashMap のキーでは、equals と hashCode が両方そろって働きます（次の表）。

SAMPLE

定義したメソッド	new X("a") を2回 add した時の size
equals だけ	2
hashCode だけ	2
両方（同じ値で一致するように）	1

hashCode が違えば、equals が true でも同じ要素とは扱われません。hashCode が同じでも equals が false なら、別の要素です。

レコードは、全成分を比べる equals と、全成分から作る hashCode を自動で持ちます。同じ成分のレコード 2 個は、HashSet や `Collectors.toSet()` で 1 個になります。equals と hashCode を書いていない普通のクラスは、同じ値を持っていても別の要素です。[12]

Set の集合演算は、`addAll` が和集合、`retainAll` が積集合、`removeAll` が差集合です。`containsAll` は部分集合の判定です。1,2,3,4 と 3,4,5 に対しては、和が [1,2,3,4,5]、積が [3,4]、差が [1,2] になります。変更が起きた時は true を返します（起きなければ false）。

`Set.of` は変更できない Set を返します。重複した要素を渡すと IllegalArgumentException になります。[10] 走査の順序は決まっておらず、将来の版で変わることもあります。

Map の操作

Map はキーから値への対応を持ち、キーの重複を許しません。

メソッド	動き	戻り値
`put(k, v)`	入れる（あれば置き換える）	前の値、前の対応が無ければ null

SAMPLE

メソッド	動き	戻り値
`get(k)`	取り出す	値、無ければ null
`getOrDefault(k, d)`	キーが無い時だけ d を返す	値か d
`putIfAbsent(k, v)`	キーに値が無い（または null）時だけ入れる	入れたら null、あれば今の値
`merge(k, v, f)`	無ければ v を入れ、あれば f の結果で置き換える	新しい値、f が null を返したら対応を削除して null
`compute(k, f)`	f(キー, 今の値) の結果を入れる	新しい値
`computeIfAbsent(k, f)`	値が無い時だけ f(キー) の結果を入れる	今の値か新しい値
`computeIfPresent(k, f)`	値がある時だけ f の結果で置き換える	新しい値、無ければ null

SAMPLE

```
Map<String,Integer> m = new HashMap<>();
m.put("a", 1)  // null
m.put("a", 2)  // 1
m.getOrDefault("zz", 0)   // 0
m.put("n", null);
m.getOrDefault("n", 9)    // null（キー n には null が対応している）
m.putIfAbsent("n", 5)     // null を返し、5 を入れる
m.putIfAbsent("a", 7)     // 2 を返し、変更しない

Map<String,Integer> c = new TreeMap<>();
for (String w : "a b a c a b".split(" ")) c.merge(w, 1, Integer::sum);
c   // {a=3, b=2, c=1}
c.merge("a", 1, (x, y) -> null)   // null を返し、a の対応を削除 →
{b=2, c=1}
c.compute("b", (k, v) -> v == null ? 1 : v * 10)   // 20
c.compute("z", (k, v) -> v == null ? 1 : v * 10)   // 1（z が新しく入
る）
c.computeIfAbsent("q", k -> 42)     // 42（q が入る）
c.computeIfAbsent("q", k -> 0)      // 42（すでにあるので関数は使われな
い）
c.computeIfPresent("nope", (k, v) -> 1)    // null（キーが無いので何も
しない）
c.computeIfPresent("q", (k, v) -> null)    // null を返し、q の対応を削
除
```

`getOrDefault` はキーが無い時にだけ既定値を返します。キーがあって値が null の時は null を返します（キーに null が対応している場合）。

SAMPLE

entrySet を走査して `setValue` すると、Map の値が書き換わります。keySet や values のビューから remove すると、Map の対応も消えます。

```
for (Map.Entry<String,Integer> e : c.entrySet())
e.setValue(e.getValue() + 100);
c.keySet().remove("b");     // c から b の対応も消える
c.replaceAll((k, v) -> v * 2);
c.forEach((k, v) -> System.out.print(k + v + ";"));
```

null の扱いは実装ごとに違います（次の表）。

Map	null のキー	null の値	変更
HashMap	入る	入る	できる
LinkedHashMap	入る	入る	できる
TreeMap	NullPointerException	入る	できる
`Map.of`	NullPointerException	NullPointerException	UnsupportedOperationException

`Map.of("a",1,"a",2)` のようにキーが重複すると、IllegalArgumentException になります。[13] LinkedHashMap は入れた順を保ちます。すでにあるキーを put し直しても、位置は変わりません（`z`, `y`, `z` と入れて `{z=3, y=2}`）。

TreeMap は NavigableMap でもあり、検索のメソッドを持ちます（キー 10, 20, 30 の例）。

SAMPLE

メソッド	返すもの	例
`firstKey()` / `lastKey()`	最小・最大のキー	10 / 30
`floorKey(k)`	k 以下で最大のキー	`floorKey(25)` は 20
`ceilingKey(k)`	k 以上で最小のキー	`ceilingKey(25)` は 30
`lowerKey(k)`	k 未満で最大のキー（無ければ null）	`lowerKey(10)` は null
`headMap(k)`	k 未満の部分	`headMap(20)` は {10=a}
`tailMap(k)`	k 以上の部分	`tailMap(20)` は {20=b, 30=c}
`descendingMap()`	逆順のビュー	{30=c, 20=b, 10=a}

Deque と Queue

Queue は先頭から取り出す入れ物で、Deque は両端で追加と削除ができる線形のコレクションです。[14] Deque の操作には2つの形があります。失敗した時に例外を投げる形と、特別な値（null か false）を返す形です。

操作	例外を投げる形	特別な値を返す形
先頭に入れる	`addFirst(e)`	`offerFirst(e)`
末尾に入れる	`addLast(e)`（Queue の `add`）	`offerLast(e)`（Queue の `offer`）
先頭を取り除く	`removeFirst()`（Queue の `remove()`）	`pollFirst()`（Queue の `poll()`）
末尾を取り除く	`removeLast()`	`pollLast()`

SAMPLE

操作	例外を投げる形	特別な値を返す形
先頭を調べる	`getFirst()`（Queue の `element()`）	`peekFirst()`（Queue の `peek()`）
末尾を調べる	`getLast()`	`peekLast()`

Queue の add と offer の違いは、容量に制限のあるキューで入らない時の動きです（offer は false を返し、add は IllegalStateException を投げる）。[15] 空の Deque に対しては、`poll`・`peek`・`pollFirst` が null を返します。`remove()`・`element()`・`pop()`・`getFirst()` は NoSuchElementException を投げます。

スタックとして使う時は、`push` と `pop` を使います。`push` は `addFirst` と、`pop` は `removeFirst` と同じです。[14] 先頭から入れて先頭から出すので、後に入れたものが先に出ます（後入れ先出し）。

```
Deque<Integer> st = new ArrayDeque<>();
st.push(1); st.push(2); st.push(3);
st          // [3, 2, 1]（先頭が左）
st.pop()    // 3
Queue<Integer> qq = new ArrayDeque<>();
qq.add(1); qq.add(2); qq.add(3);
qq.remove()  // 1（先頭から出る）→ [2, 3]
Deque<Integer> d = new ArrayDeque<>();
d.offer(1); d.offerFirst(0); d.addLast(2); d.push(-1);  // [-1, 0, 1,
2]
d.peek() // -1   d.peekLast() // 2   d.pop() // -1   d.poll() // 0
d.pollLast() // 2   → [1]
```

SAMPLE

ArrayDeque は null の要素を禁じます。[16] `offer(null)` も `push(null)` も NullPointerException です。LinkedList は null を入れられ、`ll.add(null); ll.push(null);` で size が 2 になります。PriorityQueue は、`poll` で取り出す順が並び順の小さい方からになります（5,1,4,2 を入れて 1,2,4,5 の順）。Deque の `remove(Object)` は最初に一致した要素を、`removeLastOccurrence` は最後に一致した要素を取り除きます。

走査中の変更

ArrayList のイテレータは fail-fast です。イテレータを作った後で構造を変えると、次の操作で ConcurrentModificationException を投げます（イテレータ自身の remove と add は例外）。[17] 拡張 for 文の中で `list.remove` を呼ぶと、この例外が出ます。

```
List<String> l = new ArrayList<>(List.of("a", "b", "c"));
for (String s : l) if (s.equals("a")) l.remove(s);   //
ConcurrentModificationException
for (String s : l) if (s.equals("c")) l.remove(s);   //
ConcurrentModificationException
for (String s : l) if (s.equals("b")) l.remove(s);   // 例外なしで終わ
る → [a, c]
```

最後の形で例外が出ないのは、最後から2番目の要素を消すと size が 2 になり、カーソルの位置と並ぶからです。そこで hasNext が false を返して、ループが終わります。この挙動は保証されません。API も、例外に頼るプログラムを書くなと述べています。

正しい消し方は2つです（`Iterator.remove` と `removeIf`）。

SAMPLE

```
for (Iterator<String> it = l.iterator(); it.hasNext();)
    if (it.next().equals("b")) it.remove();    // イテレータの remove
l.removeIf(s -> s.equals("a"));                // 条件に合う要素をまとめて取り除く
```

`removeIf` は条件を満たす要素を全部取り除き、1つでも取り除いたら true を返します。[18] `it.remove()` を `next()` の前に呼ぶと IllegalStateException です。ListIterator は、走査しながら `add` と `set` もできます。1,2,3,4 の走査で、2 の後に 99 を足し、3 を 33 に変えます（結果は `[1, 2, 99, 33, 4]`）。

HashMap の keySet を走査しながら新しいキーを `put` すると、次の要素へ進む時に ConcurrentModificationException になります。最後のキーを読んだ所で `put` した時は、次へ進まずにループが終わるので、例外は出ません（JDK 25 の実測）。すでにあるキーの値を `put` で置き換えるだけなら、構造の変更ではないので例外になりません。CopyOnWriteArrayList は、走査を始めた時点の写しを読みます。走査中に add しても例外は出ず、走査は元の3要素で終わります。

Sequenced Collectionsの使用 [1.3]

SequencedCollection は、出現順（encounter order）が決まっているコレクションです。両端の操作と、逆順にする操作を持ちます（`reversed()`）。[19] SequencedCollection、SequencedSet、SequencedMap は、JDK 21 の JEP 431 で加わった型です。[20]

SAMPLE

インタフェース	親	主な実装
SequencedCollection	Collection	List と Deque の全部（ArrayList、LinkedList、ArrayDeque）
SequencedSet	Set と SequencedCollection	LinkedHashSet、TreeSet（SortedSet 経由）
SequencedMap	Map	LinkedHashMap、TreeMap（SortedMap 経由）

HashSet と HashMap は出現順を持たないので、これらではありません。`new HashSet<Integer>().getFirst()` はコンパイルエラーです（Set には getFirst が無い）。PriorityQueue も SequencedCollection ではありません。

SequencedCollection のメソッド

メソッド	動き
`getFirst()` / `getLast()`	先頭と末尾の要素を返す（空なら NoSuchElementException）
`addFirst(e)` / `addLast(e)`	先頭と末尾に入れる
`removeFirst()` / `removeLast()`	先頭と末尾を取り除いて返す（空なら NoSuchElementException）
`reversed()`	逆順のビューを返す

SAMPLE

```
List<String> l = new ArrayList<>(List.of("a", "b", "c"));
l.getFirst() + l.getLast()      // "ac"
l.addFirst("z"); l.addLast("y");   // [z, a, b, c, y]
l.removeFirst(); l.removeLast();   // z と y を取り除いて [a, b, c]
```

空の ArrayList と空の Deque では、getFirst、getLast、removeFirst、removeLast のどれも NoSuchElementException を投げます。空の `List.of()` では、getFirst と getLast は NoSuchElementException、removeFirst と removeLast は UnsupportedOperationException です（JDK 25 の実測）。[19]

SequencedCollection インタフェース自身の addFirst と addLast の既定の実装は、必ず UnsupportedOperationException を投げます。[19] 子のインタフェースか実装クラスが上書きしている時だけ動きます（List は `add(0, e)` を呼ぶ既定の実装で上書きし、ArrayList・LinkedList・ArrayDeque・LinkedHashSet は自分のクラスで上書きしている）。不変の `List.of(1, 2).addFirst(0)` は UnsupportedOperationException を投げます。固定長の `Arrays.asList(1, 2)` も、`addFirst(0)` と `removeFirst()` で UnsupportedOperationException です。

reversed() の逆順のビュー

`reversed()` は逆順のコピーではなく、元のコレクションを逆向きに見るビューを返します。実装が許せば、ビューへの変更は元のコレクションに反映されます。[19]

SAMPLE

```
List<String> l = new ArrayList<>(List.of("a", "b", "c"));
List<String> rv = l.reversed();   // [c, b, a]
rv.add("NEW");                    // ビューの末尾 = 元の先頭
l    // [NEW, a, b, c]
rv   // [c, b, a, NEW]
rv.addFirst("F");                 // ビューの先頭 = 元の末尾
l    // [NEW, a, b, c, F]
l.set(0, "A");                    // 元の変更もビューから見える
rv   // [F, c, b, a, A]
List.of(1, 2).reversed().add(3);  // UnsupportedOperationException
```

ビューへの変更は元に届きます。元の変更がビューから見えるかは、API では実装次第です。ArrayList・LinkedList・ArrayDeque では、元の変更もビューから見えます（JDK 25 の実測）。元が不変なら、ビューも不変になります（変更はできない）。

SequencedSet と SequencedMap

LinkedHashSet は SequencedSet で、`addFirst` と `addLast` は動きます。すでにある要素を渡すと、その要素が先頭や末尾へ移ります。[21]

```
LinkedHashSet<String> ls = new LinkedHashSet<>(List.of("b", "a", "c"));
ls.addFirst("c");   // [c, b, a]（c が先頭へ移る）
ls.addLast("b");    // [c, a, b]（b が末尾へ移る）
ls.reversed()       // [b, a, c]
```

TreeSet は並び順で位置が決まるので、`addFirst` と `addLast` は必ず UnsupportedOperationException になります。[22] `getFirst`、`getLast`、

SAMPLE

`removeFirst`、`reversed` は動きます（`[a, b, c]` に `removeFirst()` を呼ぶと [b, c]）。

SequencedMap は、先頭と末尾の対応を扱うメソッドを足します。

メソッド	動き
`firstEntry()` / `lastEntry()`	先頭と末尾の対応（空なら null）
`pollFirstEntry()` / `pollLastEntry()`	取り除いて返す（空なら null）
`putFirst(k, v)` / `putLast(k, v)`	入れて（あれば値を置き換えて）先頭・末尾に置く
`reversed()`	逆順のビュー
`sequencedKeySet()` / `sequencedValues()` / `sequencedEntrySet()`	キー・値・対応の逆順にできるビュー

SAMPLE

```
LinkedHashMap<String,Integer> lm = new LinkedHashMap<>();
lm.put("x", 1); lm.put("y", 2); lm.put("z", 3);
lm.firstEntry() + " " + lm.lastEntry()   // x=1 z=3
lm.putFirst("w", 0);    // {w=0, x=1, y=2, z=3}
lm.putLast("x", 10);    // {w=0, y=2, z=3, x=10}（x が末尾へ移り、値も10 に変わる）
lm.putFirst("z", 33);   // {z=33, w=0, y=2, x=10}
lm.pollFirstEntry() + " " + lm.pollLastEntry()   // z=33 x=10 → lm は{w=0, y=2}
lm.reversed()           // {y=2, w=0}
lm.sequencedKeySet()    // [w, y]
lm.sequencedKeySet().remove("y");   // lm からも y が消える
new LinkedHashMap<String,Integer>().firstEntry()   // null
```

`firstEntry`・`lastEntry`・`pollFirstEntry`・`pollLastEntry` が返す Entry は、呼び出した時点の写しです。この Entry に `setValue` を呼ぶと UnsupportedOperationException になります。[23] TreeMap は並び順で位置が決まるので、`putFirst` と `putLast` は使えません（UnsupportedOperationException）。TreeMap でも `firstEntry`、`pollFirstEntry`、`reversed`、`sequencedKeySet` は使えます。`Collections.unmodifiableSequencedMap` で包んだものの `putFirst` も UnsupportedOperationException です。

LinkedHashMap をアクセス順（`new LinkedHashMap<>(16, 0.75f, true)`）で作ると、`get` で触ったキーが末尾へ回ります。a, b を入れて `get("a")` すると、`{b=2, a=1}` になります（キー a が末尾へ移る）。

SAMPLE

総称型の使用 [1.4]

総称型（ジェネリクス）は、型を引数のように後から決められるクラスやメソッドです。クラスが1つ以上の型変数を宣言すると、総称クラスになります。[24] 型パラメータは `<T>` の形で書き、使う側が型引数を渡します（`List<String>` など）。

宣言と型引数の推論

```
class Box<T> {                          // 総称クラス
    T v; Box(T v) { this.v = v; }
    T get() { return v; }
    <R> Box<R> map(Function<T,R> f) { return new Box<>(f.apply(v)); }
// 総称メソッド
}
interface Pair<K, V> { K k(); V v(); }   // 総称インタフェース（型パラメータは複数）

Box<String> b = new Box<>("hello");      // ダイヤモンド演算子
b.map(String::length).get()              // 5
List<String> list = new ArrayList<>();   // 右辺の型引数は左辺から推論される
Util.<Integer>max(1, 5);                 // 型引数を明示することもできる
```

new の型引数を `<>` で省き、コンパイラに推論させる形をダイヤモンドと呼びます。[25] `var list = new ArrayList<>();` は左辺に型が無いので、推論は `ArrayList<Object>` になり、文字列も整数も入ります。型引数に基本データ型

SAMPLE

は使えません。`List<int>` はコンパイルエラーです（required: reference, found: int）。総称メソッド `static <T> void swap(List<T> l, int i, int j)` は、呼び出し側の引数から T を推論します。戻り値の型に T を使う総称メソッドでは、引数と代入先の両方から T が推論されます。`static <T> List<T> pair(T a, T b)` に `pair("x", 2)` を渡すと、`List<Object>` には代入できますが、`List<String>` に代入するとコンパイルエラーです（2 の Integer が String ではない）。

境界のある型パラメータ

型変数の境界は `extends` の後に型を書きます。`&` でインタフェースを続けて複数の境界にできます。[26] 境界を書かない型変数の境界は Object です。

```
static <T extends Comparable<T>> T max(T x, T y) { return
x.compareTo(y) >= 0 ? x : y; }
static <T extends Number & Comparable<T>> T maxN(List<T> l) { ... }
// クラスは先頭に1つ、インタフェースは & で続ける

max("a", "b")     // "b"
max(3, 2)         // 3
maxN(List.of(3, 9, 4))   // 9
max(new Object(), new Object())   // コンパイルエラー（Object は
Comparable ではない）
class A<T extends Number> {}
A<String> x;              // コンパイルエラー（String は上限 Number の範
囲外）
```

SAMPLE

境界のおかげで、T の値に対して `compareTo` や `doubleValue` を呼べるようになります（Object には無いメソッド）。`T extends Number & Comparable<T>` では、クラスの境界（Number）を最初に書き、インタフェースを後ろへ `&` でつなぎます。

`List<Number>` には、Integer・Double・Long など Number の子の型の値を入れられます。入れた値は元の型のまま残るので、7・0.5・9L を入れたリストの表示は `[7, 0.5, 9]` です（9 は Long のまま）。

ワイルドカード

型引数に `?` を使うと、型パラメータについて一部しか知らなくてよい場面を書けます。[27] 上限は `? extends T`、下限は `? super T` で付けます。

書き方	意味	読む（get）	書く（add）
`List<? extends Number>`	Number かその子の型のリスト	Number として読める	null 以外は入れられない
`List<? super Integer>`	Integer かその親の型のリスト	Object としてしか読めない	Integer を入れられる
`List<?>`	どんな型でもよいリスト	Object として読める	null 以外は入れられない

SAMPLE

```
static double sum(List<? extends Number> l) { double s = 0; for (Number
n : l) s += n.doubleValue(); return s; }
static void fill(List<? super Integer> l) { l.add(1); l.add(2); }

sum(List.of(1, 2)); sum(List.of(1.5));     // どちらも通る
fill(new ArrayList<Number>()); fill(new ArrayList<Object>());   // どち
らも通る
List<? extends Number> ext = new ArrayList<Integer>();
ext.add(1);                 // コンパイルエラー（何のリストか分からな
いので入れられない）
Number n = ext.get(0);      // 通る
List<? super Integer> sup = new ArrayList<Number>();
sup.add(5);                 // 通る
Integer i = sup.get(0);     // コンパイルエラー（Object としてしか読め
ない）
```

取り出すだけなら extends、入れるだけなら super と覚えます。取り出しも入れもする時は、ワイルドカードを使わず型変数で書きます（`<T> void f(List<T> l)`）。

`List<String>` は `List<Object>` の部分型ではありません。`List<Integer> li = ...; List<Number> ln = li;` はコンパイルエラーです。配列は違い、`Object[] oa = new String[1]; oa[0] = 1;` はコンパイルが通り、実行時に ArrayStoreException になります。ワイルドカードの型どうしは、包含の規則で関係が決まります。`List<Integer>` は `List<? extends Number>` に代入できます。[27]

SAMPLE

型消去の制約

コンパイラは、型引数を確かめた後に消します。これが型消去で、パラメータ化された型を、型引数を除いた型に写します。[28] 実行時の型は、`List<String>` も `List<Integer>` も同じ `List` です（getClass() の比較は true）。型消去のせいで、型引数を使う次の書き方が制限されます。

書いたもの	結果	理由
`o instanceof List<String>`（o が Object）	コンパイルエラー（Object cannot be safely cast to List）	実行時に型引数が残らない
`o instanceof List<?>`	通る	型引数の情報は要らない
`c instanceof List<String>`（c が `Collection<String>`）	通る	型引数を保ったまま安全にできる
`new T()`	コンパイルエラー（unexpected type）	実行時に T が分からない
`new T[3]`、`new List<String>[3]`	コンパイルエラー（generic array creation）	配列は実行時の型を持つ
`static T x;` と `static void f(T t)`	コンパイルエラー（non-static type variable T cannot be referenced from a static context）	T はインスタンスごとに決まる
`void f(List<String> l)` と `void f(List<Integer> l)`	コンパイルエラー（消去後の署名が同じ・JLS 8.4.8.3）[29]	消去後は同じ `f(List)`
`o instanceof T`	コンパイルエラー（Object cannot be safely cast to T）	実行時に T が分からない

SAMPLE

型引数を持つ型の多くは、実行時に型の情報を全部は持ちません（reifiableでない型）。[30] instanceof の右辺が reifiable でない型の時は、左辺の型から未検査にならずにキャストできなければ、コンパイルエラーです。[31] `T[] arr = (T[]) new Object[1];` は、キャストが未検査の警告になるだけで、コンパイルは通ります。

型引数を付けない総称型の名前を raw 型と呼びます。[32] raw 型を使うと、未検査の警告が出ます（`-Xlint:all` で表示）。raw 型を通すと、型の検査をすり抜けて、後で読む場所で ClassCastException になります。

```
List<String> ls = new ArrayList<>();
List raw = ls;          // 警告: rawtypes
raw.add(42);            // 警告: unchecked。ここでは例外にならない
ls.size();              // 1（Integer が String のリストに入っている）
String s = ls.get(0);   // ClassCastException（読む時に String へのキャストが入る）
```

警告はコンパイルを止めません。

Comparatorインタフェースを使用したコレクションのソート [1.5]

Comparable と Comparator

Comparable を実装したクラスの順序は、そのクラスの自然順序です。[33] `compareTo` は、自分が相手より小さい・等しい・大きいに応じて、負・0・正の値を返します。Comparator の `compare(a, b)` は、第1引数が小さい・等しい・大きいに応じて、負・0・正を返します。[34]

SAMPLE

	Comparable	Comparator
メソッド	`int compareTo(T o)`	`int compare(T a, T b)`
どこに書く	比べられるクラスの中	別のクラスやラムダ
順序の数	1つ（自然順序）	いくつでも作れる

```
"b".compareTo("a")        // 1
"a".compareTo("c")        // -2
"apple".compareTo("app")  // 2（長さの差）
"B".compareTo("a")        // -31（大文字は小文字より前）
Integer.valueOf(5).compareTo(3)   // 正の値（JDK 25 では 1）
```

返す値の大きさに意味はありません。符号だけが意味を持つので、`a - b` で比べると桁あふれで順序が逆転します。`Integer.MAX_VALUE` と -1 を `x - y` で比べると -2147483648 になります（`Integer.compare` なら正の値）。

TreeSet と TreeMap は自然順序か、作る時に渡した Comparator で並べます。[35] 重複の判定も、equals ではなくこの比較が 0 かどうかで決まります。

```
class Loose implements Comparable<Loose> { public int compareTo(Loose
o) { return 0; } ... }
TreeSet<Loose> tl = new TreeSet<>();   // 2つ入れても size は 1
Set<String> t2 = new TreeSet<>(Comparator.comparing(String::length));
t2.addAll(List.of("aa", "bb", "c"));   // [c, aa]（bb は長さが aa と同
じなので捨てられる）
new TreeSet<>(Comparator.reverseOrder())   // 大きい順
```

SAMPLE

TreeSet は `first`、`last`、`headSet`（未満）、`ceiling`（以上で最小）、`descendingSet` なども持ちます。「未満」「以上」「最小」は、その TreeSet の並び順（Comparator）で決まり、数の大小では決まりません。`new TreeSet<>(Comparator.reverseOrder())` に 10・20・30 を入れると `[30, 20, 10]` で、`headSet(20)` は `[30]`、`ceiling(25)` は 20 です。[36] Comparable を実装しない型を TreeSet に入れると、1つ目から ClassCastException になります（Comparator を渡していない場合）。

Comparator の組み立て

メソッド	動き
`Comparator.comparing(f)`	キーを取り出す関数 f から、そのキーで比べる Comparator を作る
`comparingInt(f)`	int のキーで比べる（ボクシングが要らない）
`thenComparing(c)`	前の Comparator で等しい時だけ次の Comparator で比べる
`reversed()`	この Comparator の逆の順序の Comparator
`naturalOrder()` / `reverseOrder()`	自然順序とその逆
`nullsFirst(c)` / `nullsLast(c)`	null を null でない値より小さい・大きいとみなす

SAMPLE

```
record P(String name, int age) {}
// 並び替え前: bob30, amy25, cat30, al25
ps.sort(Comparator.comparing(P::name));
// al, amy, bob, cat
ps.sort(Comparator.comparingInt(P::age).thenComparing(P::name));
// al25, amy25, bob30, cat30
ps.sort(Comparator.comparingInt(P::age).thenComparing(P::name).reversed
());   // cat30, bob30, amy25, al25
ps.sort(Comparator.comparingInt(P::age).reversed().thenComparing(P::nam
e));   // bob30, cat30, al25, amy25
ps.sort(Comparator.comparing(P::name, Comparator.reverseOrder()));
// cat, bob, amy, al
```

`reversed()` は、それまでに組み立てた全体を逆にします。3行目は年齢も名前も逆順で、4行目は年齢だけが逆順です（同じ年齢の中は名前の昇順）。reversed を置く位置で、結果が変わります。

ラムダの型を明示しないと、推論が途切れます。

```
ps.sort(Comparator.comparing(p -> p.name()).reversed());        // コン
パイルエラー（p が Object になる）
ps.sort(Comparator.comparing((P p) -> p.name()).reversed());    // 型を
明示すれば通る
```

`reversed()` の受け手になった `comparing(…)` の呼び出しは、代入や引数の文脈に無いので、ポリ式ではありません。代入先の型を使わずに `comparing`

SAMPLE

の型引数を推論するので、p は Object とみなされ、`name()` が見つからなくなります。メソッド参照（`P::name`）か、ラムダに型を書けば避けられます。

null を含むリストでは、null の位置を nullsFirst と nullsLast が決めます（`naturalOrder()` だけだと NullPointerException）。

```
List<String> wn = new ArrayList<>(Arrays.asList("b", null, "a"));
wn.sort(Comparator.nullsFirst(Comparator.naturalOrder()));   // [null, a, b]
wn.sort(Comparator.nullsLast(Comparator.reverseOrder()));    // [b, a, null]
wn.sort(Comparator.naturalOrder());     // NullPointerException
```

文字列の自然順序は文字コードの順なので、大文字が小文字より先に並びます。並べ替えは安定で、等しい要素は元の順を保ちます。

["banana","Apple","cherry","apple","Banana"] への並べ方	結果
`Collections.sort(w)`	`[Apple, Banana, apple, banana, cherry]`
`w.sort(String.CASE_INSENSITIVE_ORDER)`	`[Apple, apple, banana, Banana, cherry]`
`w.sort(Comparator.reverseOrder())`	`[cherry, banana, apple, Banana, Apple]`
`w.sort(Comparator.comparing(String::length))`	`[Apple, apple, banana, cherry, Banana]`

SAMPLE

["banana","Apple","cherry","apple","Banana"] への並べ方	結果
長さの次に自然順序（`comparing(String::length).thenComparing(Comparator.naturalOrder())`）	`[Apple, apple, Banana, banana, cherry]`

List.sort・Collections.sort と ClassCastException

`Collections.sort(List)` は、全要素が Comparable を実装していなければなりません。[37] 静的に型が分かる場合は、コンパイルの段階で止まります。

```
class Foo {}
List<Foo> l = new ArrayList<>();
Collections.sort(l);     // コンパイルエラー（Foo は Comparable ではない）
l.sort(null);            // コンパイルは通る。Foo の要素が入っていれば実行時に ClassCastException
```

`List.sort(null)` は、Comparator に null を渡すと、自然順序で並べます。この時、リストの全要素が Comparable を実装していなければなりません。[38] 要素どうしを比べられなければ、ClassCastException が出ます。raw 型や `List<Object>` を経由すると、コンパイルを通り抜けて、実行時の例外になります。

実行	結果
`new TreeSet<Object>().add(new Foo())`	ClassCastException

SAMPLE

実行	結果
`new TreeMap<Object,Integer>().put(new Foo(), 1)`	ClassCastException
1 と "a" の混在リストを `Collections.sort`	ClassCastException
Foo が2つのリストを `sort((x, y) -> 0)`	例外なし（Comparator で比べる）
`ps.sort(null)`（P は Comparable ではない、4要素）	ClassCastException

Comparator を渡さない TreeSet と TreeMap では、入れる要素（キー）はすべて Comparable を実装していなければなりません。実装していなければ、1つ目の要素を入れる時にも ClassCastException です。リストの sort に Comparator を渡すと、全要素はその Comparator で比べられればよく、Comparable の実装は要りません。[38]

コレクション用の便利なメソッドの使用 [1.6]

Collections の便利なメソッド

メソッド	動き
`sort(list)` / `sort(list, c)`	並べ替える（自然順序か c）
`reverse(list)`	要素の順を逆にする
`shuffle(list)` / `shuffle(list, rnd)`	並びを無作為に入れ替える
`swap(list, i, j)` / `rotate(list, d)`	2つの位置を入れ替える／d だけ回す

SAMPLE

メソッド	動き
`max(c)` / `min(c)` と `max(c, cmp)`	最大・最小（自然順序か cmp）。空なら NoSuchElementException
`frequency(c, o)`	o と等しい要素の数
`binarySearch(list, key)`	並べてあるリストから探す
`unmodifiableList(list)`	変更できないビュー
`nCopies(n, o)` / `emptyList()` / `singletonList(o)`	変更できない固定のリスト
`addAll(c, ...)` / `fill(list, o)`	まとめて足す／全部同じ値にする

```
List<Integer> n = new ArrayList<>(List.of(1, 2, 3, 4, 5));
Collections.reverse(n);      // [5, 4, 3, 2, 1]
Collections.swap(n, 0, 4);   // [1, 4, 3, 2, 5]
Collections.rotate(n, 2);    // [2, 5, 1, 4, 3]（右へ2つ回す）
Collections.max(List.of(3, 9, 4))                                    // 9
Collections.max(List.of("aa", "b", "ccc"),
Comparator.comparing(String::length))   // "ccc"
Collections.min(List.of(3, 9, 4), Comparator.reverseOrder())       // 9
（逆順での最小 = 最大）
Collections.frequency(List.of(1, 2, 1, 1), 1)   // 3
Collections.frequency(List.of("a"), null)       // 0
Collections.binarySearch(List.of(1,2,3,4,5), 4)   // 3
Collections.binarySearch(List.of(1,2,3,4,5), 10)  // -6
```

SAMPLE

`Collections.frequency` は、指定したオブジェクトと等しい要素の数を返します。`Collections.max` は自然順序で最大の要素を返します。[39] 不変のリストを `Collections.sort` に渡すと UnsupportedOperationException です（`List.of(3,1,2)` の例）。`emptyList()` や `nCopies` への add も UnsupportedOperationException になります。

`Collections.unmodifiableList` は、変更できない「ビュー」を返します。元のリストの変更は、ビューから見えます。[39]

```
List<Integer> src = new ArrayList<>(List.of(1, 2));
List<Integer> view = Collections.unmodifiableList(src);
src.add(3);
view         // [1, 2, 3]（元の変更が見える）
view.add(4); // UnsupportedOperationException
```

不変のコレクションの作り方

作り方	元との関係	null	変更
`List.of(...)`、`Set.of(...)`、`Map.of(...)`	値から作る	入れると NullPointerException	UnsupportedOperationException
`List.copyOf(c)`、`Set.copyOf(c)`、`Map.copyOf(m)`	コピー（元とは切れる）	元に null があると NullPointerException	UnsupportedOperationException
`Collections.unmodifiableList(l)` など	ビュー（元とつながる）	入っていてもよい	ビューからはできない

SAMPLE

`List.copyOf` は、与えたコレクションの要素を、走査の順に持つ不変のリストを返します。[40] `Map.copyOf` は与えた Map の対応を持つ不変の Map を返します。元に変更を加えても、コピーには届きません。

```
List<Integer> src = new ArrayList<>(List.of(1, 2, 3));
List<Integer> cp = List.copyOf(src);
src.add(4);
cp      // [1, 2, 3]
List.copyOf(cp) == cp    // JDK 25 では true（API は、不変のリストなら普通はコピーを作らないと書く）
List.copyOf(src) == src  // false
List.copyOf(Arrays.asList(1, null));   // NullPointerException
Collections.unmodifiableList(new ArrayList<>(Arrays.asList(1, null)))
// [1, null]（null が入っていてよい）
Set.copyOf(List.of(1, 1, 2)).size()    // 2（重複は1つにまとまる。Set.of と違い例外にならない）
```

`Map.of` の重複キーと `Set.of` の重複は、例外になります。`Set.copyOf` は重複を許します。

Arrays のメソッドと Collection のデフォルトメソッド

メソッド	動き
`Arrays.asList(...)`	配列に支えられた固定長のリスト
`Arrays.sort(a)` / `sort(a, from, to)` / `sort(a, cmp)`	並べ替える（`from` を含み `to` を含まない）
`Arrays.binarySearch(a, key)`	並べてある配列から探す

SAMPLE

メソッド	動き
`Arrays.equals(a, b)`	長さと要素が全部等しいか
`Arrays.compare(a, b)`	辞書式の比較（負・0・正）
`Arrays.mismatch(a, b)`	最初に違う添字、違いが無ければ -1
`Arrays.deepEquals`、`deepToString`	入れ子の配列を中身まで比べる・文字にする
`Arrays.fill`、`copyOf`、`copyOfRange`	埋める・長さを変えて写す・範囲を写す

```
int[] x = {5, 3, 9, 1};
Arrays.sort(x);                          // [1, 3, 5, 9]
Arrays.binarySearch(x, 9)                // 3
Arrays.binarySearch(x, 4)                // -3（挿入点 2 → -(2) - 1）
Arrays.binarySearch(x, 0)                // -1（挿入点 0）
Arrays.binarySearch(x, 100)              // -5（挿入点 4）
x.equals(new int[]{1,3,5,9})             // false（配列の equals は参照の比較）
Arrays.equals(x, new int[]{1,3,5,9})     // true
Arrays.compare(new int[]{1,2,3}, new int[]{1,2,4})   // -1
Arrays.compare(new int[]{1,2}, new int[]{1,2,0})     // -1（短い方が先）
Arrays.compare(new int[]{3}, new int[]{1,2})         // 1
Arrays.mismatch(new int[]{1,2,3}, new int[]{1,2,4})  // 2
Arrays.mismatch(new int[]{1,2}, new int[]{1,2})      // -1
Arrays.mismatch(new int[]{1,2}, new int[]{1,2,3})    // 2（短い方の長さ）
```

SAMPLE

binarySearch は、検索の前に配列が並べてあることが前提で、並べていなければ結果は決まりません。見つからない時は `(-(挿入点) - 1)` を返します。[41] 挿入点は、キーを入れるとしたら入る位置です（挿入点が 2 なら -3 を返す）。

二次元の配列は、`equals` が中の配列を参照で比べるので、中身まで比べる deep の付くメソッドを使います。

式（ int[][] d1 = {{1,2},{3}}, d2 = {{1,2},{3}}; ）	結果
`d1.equals(d2)`	false
`Arrays.equals(d1, d2)`	false（中の配列は参照で比べる）
`Arrays.deepEquals(d1, d2)`	true
`Arrays.toString(d1)`	`[[I@` で始まる文字列（ハッシュ値は実行ごとに変わる）
`Arrays.deepToString(d1)`	`[[1, 2], [3]]`

SAMPLE

```
int[] r = {9, 8, 7, 6, 5};
Arrays.sort(r, 1, 4);    // [9, 6, 7, 8, 5]
Arrays.sort(r, 3, 1);    // IllegalArgumentException (fromIndex(3) > toIndex(1))
Arrays.sort(r, 0, 9);    // ArrayIndexOutOfBoundsException
Integer[] bx = {3, 1, 2};
Arrays.sort(bx, Comparator.reverseOrder());   // [3, 2, 1]
Arrays.sort(new int[]{3,1}, Comparator.reverseOrder());   // コンパイルエラー（int[] に Comparator は使えない）
String[] sa = {"b", "B", "a", "A"};
Arrays.sort(sa);         // [A, B, a, b]
```

Comparator を渡せるのは、参照型の配列です。基本データ型の配列は、コンパイルエラーになります（`int[]` は不可）。null の配列を渡すと NullPointerException です。

Collection と List のデフォルトメソッドは、ラムダを受け取る形です（次の表）。

メソッド	動き
`Iterable.forEach(action)`	全要素に処理を行う（Map には Map 自身の `forEach((k, v) -> ...)`）
`Collection.removeIf(predicate)`	条件を満たす要素を全部取り除く。取り除けば true
`List.replaceAll(operator)`	各要素を演算子の結果で置き換える（Map にも `replaceAll((k, v) -> ...)`）

SAMPLE

```
List<Integer> l = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));
l.removeIf(v -> v % 2 == 0)   // true → [1, 3, 5]
l.removeIf(v -> v > 100)      // false（取り除いていない）
l.replaceAll(v -> v * 10);    // [10, 30, 50]
l.forEach(System.out::print); // 103050
Map<String,Integer> m = new TreeMap<>(Map.of("a", 1, "b", 2));
m.values().removeIf(v -> v == 1);                          // {b=2}
m.entrySet().removeIf(e -> e.getKey().equals("b"));        // {}
l.replaceAll(null)            // NullPointerException
```

`Iterable.forEach` は、全要素の処理が終わるか例外が出るまで続きます。Map は Iterable ではないので、`m.forEach(...)` は Map 側の `forEach(BiConsumer)` です。`List.of(1).replaceAll(v -> v)` と `List.of(3,1,2).sort(null)` は UnsupportedOperationException です。`Arrays.asList` の固定長リストは、サイズを変えない操作なら動きます（`replaceAll` と `Collections.sort`）。

`subList` はビューで、`list.subList(1, 3).set(0, 99)` は元のリストを書き換えます（`[1,2,3,4]` なら `[1, 99, 3, 4]`）。`[1,2,3,4]` から `subList(1,3)` を clear すると `[1, 4]` になります（元の範囲が消える）。元のリストを構造的に変えると、それ以前に作ったイテレータの `next()` は ConcurrentModificationException を投げます。

List と Set の equals は、実装の種類に関係なく中身で決まります。`List.of(1,2).equals(Arrays.asList(1,2))` は true です。`new HashSet<>(List.of(1,2)).equals(Set.of(2,1))` も true です。List の hashCode は要素から決まります（`List.of(1,2).hashCode()` は 994）。`toArray()` は `Object[]`

SAMPLE

を返します。型を指定する `toArray(new Integer[0])` と `toArray(Integer[]::new)` は、`Integer[]` を返します。

SAMPLE

サンプルはここまでです

続き(第2章〜巻末)は、Java Gold SE 25 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

Footnotes

1. https://docs.oracle.com/javase/specs/jls/se25/html/jls-5.html#jls-5.1.7 ↩
2. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Integer.html#equals(java.lang.Object) ↩
3. Java SE 25 API, java.util.Set.contains・java.util.Map.get（引数は Object ）。 https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Set.html#contains(java.lang.Object) ↩
4. https://docs.oracle.com/javase/specs/jls/se25/html/jls-5.html#jls-5.1.8 ↩
5. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Integer.html#valueOf(int) ↩
6. https://docs.oracle.com/javase/specs/jls/se25/html/jls-15.html#jls-15.12.2 ↩
7. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/List.html ↩

SAMPLE

8. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/List.html#unmodifiable ↩ ↩2
9. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html#asList(T...) ↩ ↩2
10. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Set.html ↩ ↩2
11. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedHashSet.html ↩
12. Java SE 25 API, java.lang.Record.equals・hashCode。https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Record.html#equals(java.lang.Object) ↩
13. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Map.html#unmodifiable ↩
14. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Deque.html ↩ ↩2
15. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Queue.html ↩
16. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayDeque.html ↩
17. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayList.html ↩

SAMPLE

18. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Collection.html#removeIf(java.util.function.Predicate) ↩

19. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/SequencedCollection.html ↩ ↩2 ↩3 ↩4

20. https://openjdk.org/jeps/431 ↩

21. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedHashSet.html#addFirst(E) ↩

22. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeSet.html#addFirst(E) ↩

23. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/SequencedMap.html ↩

24. https://docs.oracle.com/javase/specs/jls/se25/html/jls-8.html#jls-8.1.2 ↩

25. https://docs.oracle.com/javase/specs/jls/se25/html/jls-15.html#jls-15.9 ↩

26. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html#jls-4.4 ↩

27. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html#jls-4.5.1 ↩ ↩2

28. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html#jls-4.6 ↩

SAMPLE

29. Java 言語仕様 SE 25, 8.4.8.3 オーバーライドと隠蔽の要件（消去後の署名が同じ）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-8.html#jls-8.4.8.3 ↩

30. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html#jls-4.7 ↩

31. https://docs.oracle.com/javase/specs/jls/se25/html/jls-15.html#jls-15.20.2 ↩

32. https://docs.oracle.com/javase/specs/jls/se25/html/jls-4.html#jls-4.8 ↩

33. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Comparable.html ↩

34. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Comparator.html ↩

35. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeSet.html ↩

36. Java SE 25 API, java.util.TreeSet（Comparator による順序）。https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeSet.html ↩

37. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Collections.html#sort(java.util.List) ↩

38. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/List.html#sort(java.util.Comparator) ↩ ↩2

SAMPLE

39. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Collections.html ↩ ↩2

40. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/List.html#copyOf(java.util.Collection) ↩

41. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html ↩

SAMPLE

Java Gold SE 25(1Z0-828-JPN) 学習用テキスト(無料サンプル)

版

2026年11月版(Oracle公式 Java SE 25 Programmer II(1Z0-828-JPN) 試験内容チェックリスト 準拠)

発行

IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ

info@shikaku-doujou.com

Oracle、Java、および MySQL は、Oracle Corporation およびその子会社、関連会社の米国およびその他の国における登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、Oracle との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。

SAMPLE