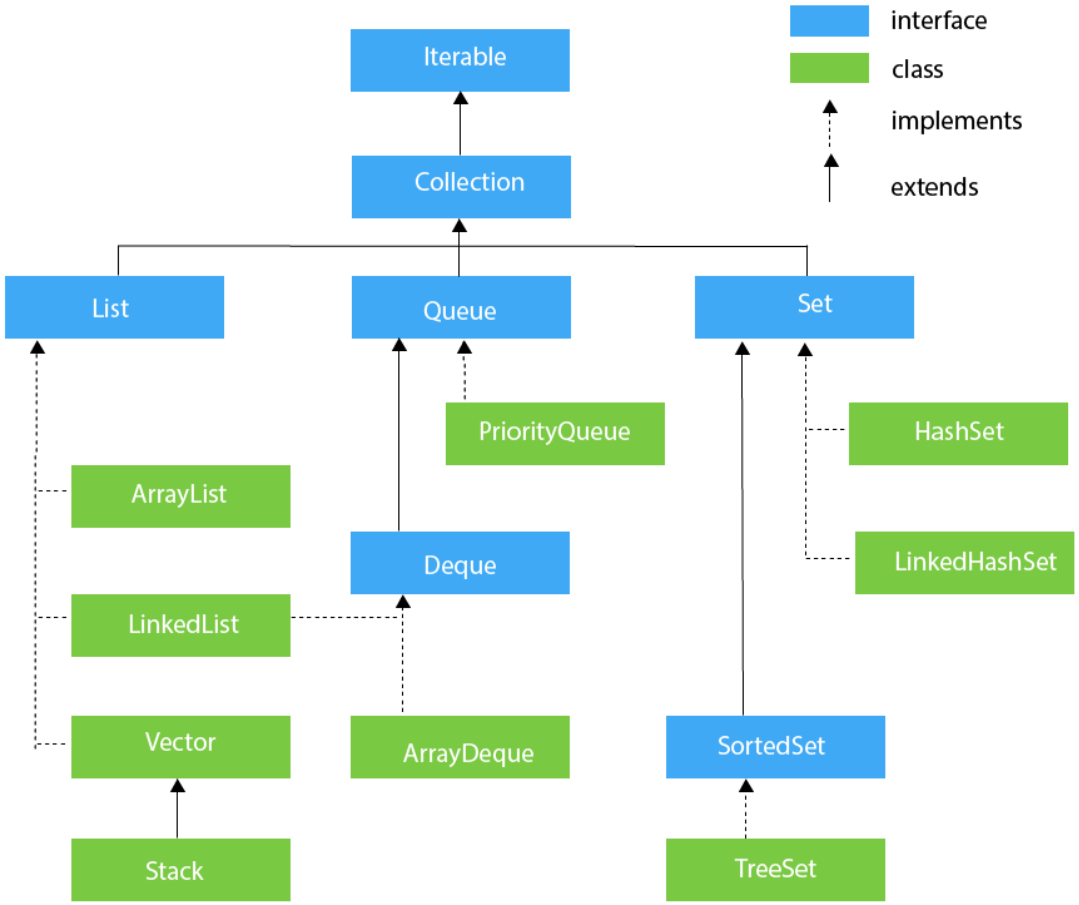
Generic classes and methods. Collection frameworks: list & map

- By Sai Teja Polisetty

Collections

- **Collection Framework** = interfaces + implementations for working with groups of objects.
- **Core interfaces: List**, **Set**, **Queue/Deque** (all extend **Collection**); **Map** is separate.
- **Benefits:** type-safe generics, ready-made algorithms, consistent APIs.
- **PTR:** "**Lists** keep order, **Sets** keep uniqueness, **Queues** handle workflow, **Maps** map keys → values."



Why Generic?

- Before Generics, Java collections like ArrayList or HashMap could store any type of object, everything was treated as an Object. It had some problems.
- If you added a String to a List, Java didn't remember its type. You had to manually cast it when retrieving. If the type was wrong, it caused a runtime error.
- With Generics, you can specify the type the collection will hold like ArrayList<String>. Now, Java knows what to expect and it checks at compile time, not at runtime.
- "imagine a 'box' that can hold *anything*. convenient, but risky."

Without Generic	With Generics

List list = new ArrayList();
 	
// raw type
list.add("42");
Integer n = (Integer) list.get(0); // ClassCastException at runtime
	List<Integer> nums = new ArrayList<>();
	nums.add(42);
	Integer n2 = nums.get(0);
	
	// no cast, type-safe

"with generics, we move the error to compile time (best place to catch it)."

Generic class

- A generic class is like a regular class but uses type parameters (like <T>).
- It can accept one or more types, making the class reusable for different data types. Such classes are called parameterized classes.

Class with single type parameter

```
class Test<T> {

    T obj;
    Test(T obj) {
        this.obj = obj;
    }
    public T getObject() { return this.obj; }
}

class Dpoint {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new
Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new
Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

Class with multiple type parameter

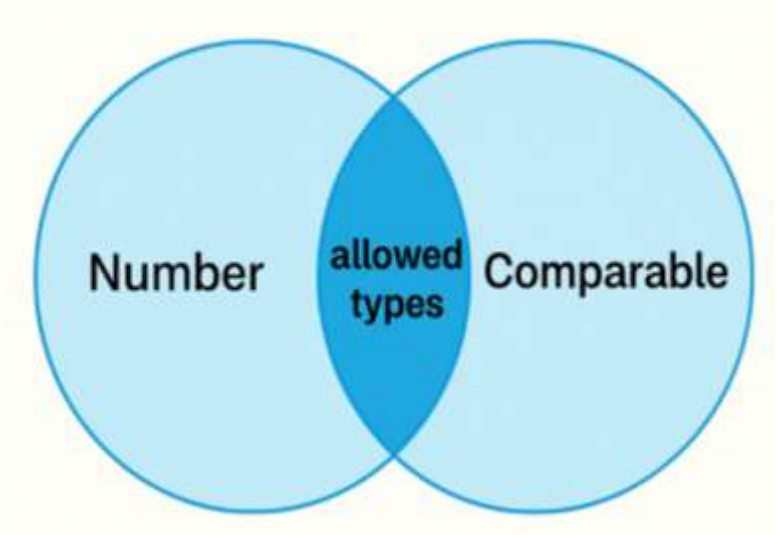
```
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

class Dpoint
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG",
15);

        obj.print();
    }
}
```

Generic Method

- A generic method is a method that can work with different data types using a type parameter. It lets you write one method that works for all types, instead of repeating the same logic.
- class Geeks {

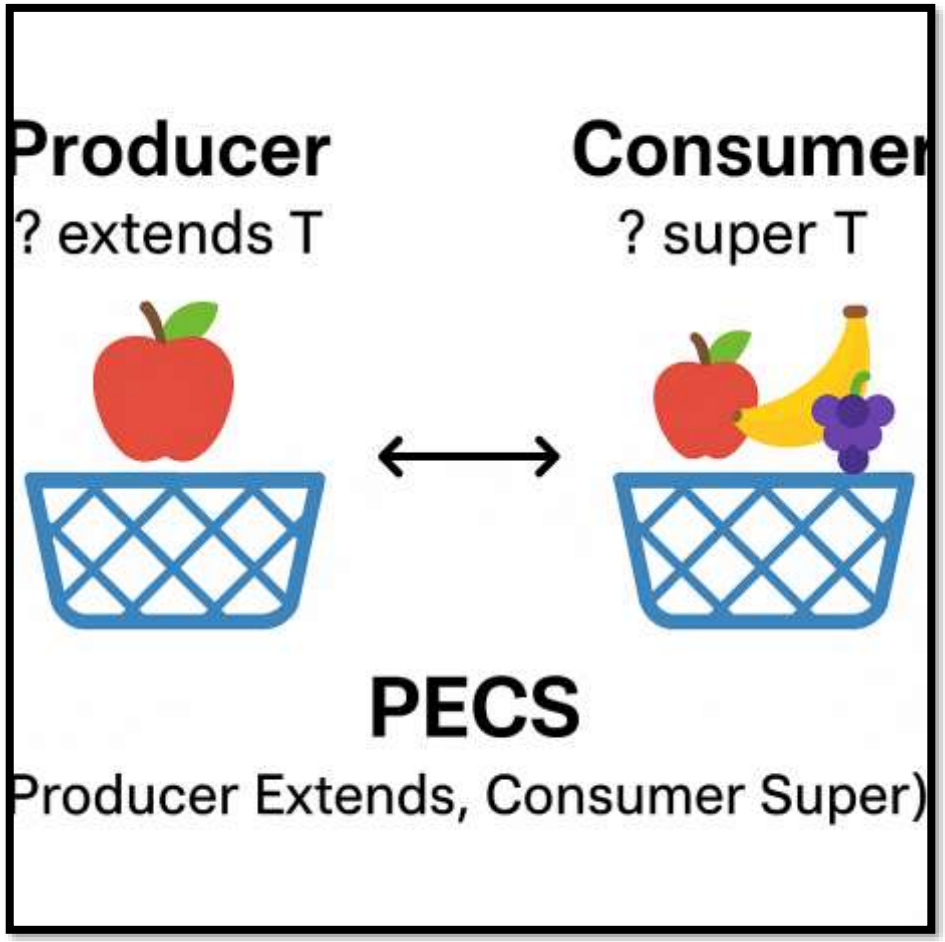
```
    // A Generic method example
    static <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()
                        + " = " + element);
    }

    public static void main(String[] args)
    {
        // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

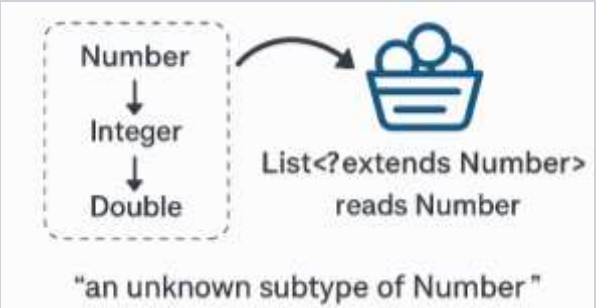
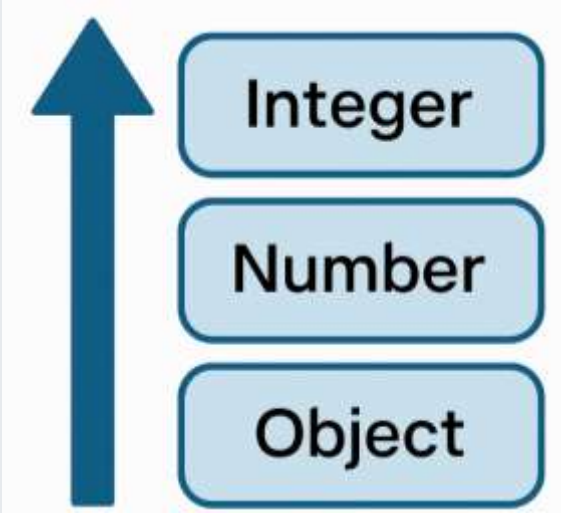
Bounded type params



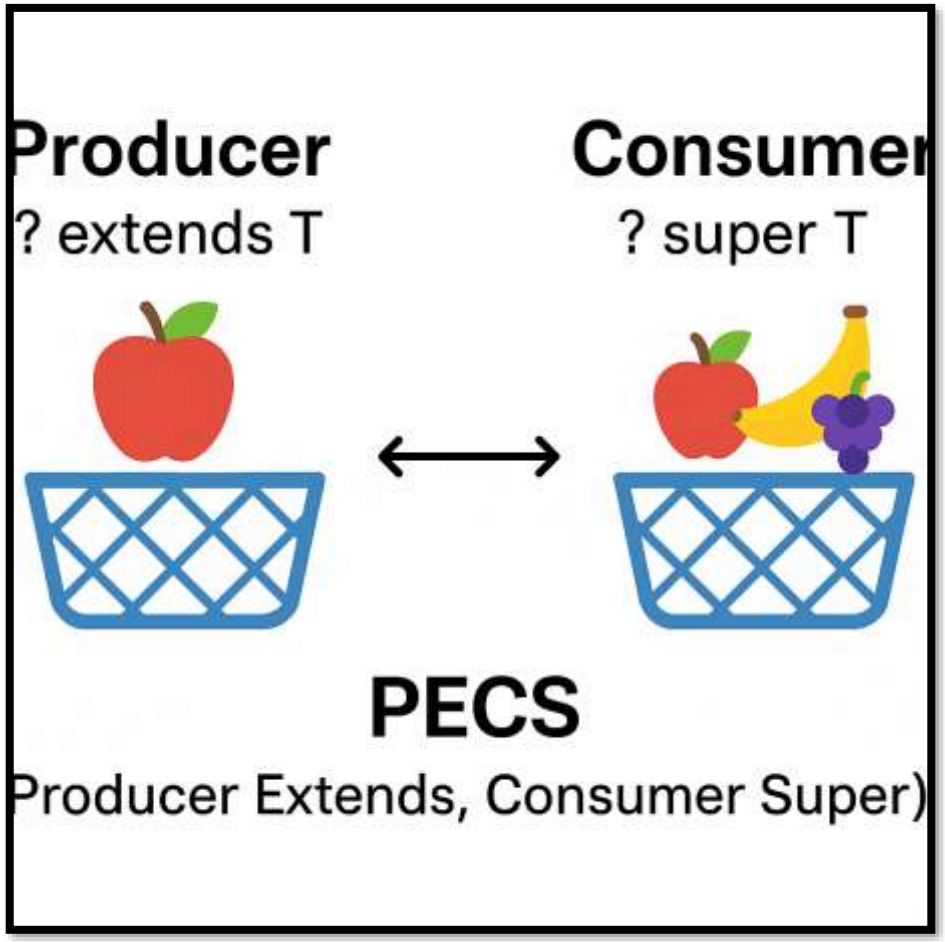
- limit which types are allowed.”
- // only numbers
  static <T extends Number> double sum(T a, T b){
  return a.doubleValue() + b.doubleValue();
  }
  // multiple bounds
  static <T extends Number & Comparable<T>> T max2(T a, T b){
  return (a.compareTo(b) >= 0) ? a : b;
  }

Producer Extends Consumer Super (PECS):



Cat.	Definition	Diagram
Extend	? extends T means “an unknown type that is T or a subtype of T.” **Inheritance direction downward**	
Super	? super T means: **“some unknown type that is T or a supertype of T.”** Example: ? super Integer could be: Integer itself Number (parent of Integer) Object (parent of everything in Java)	

Producer Extends Consumer Super (PECS):



Feature	? extends T (Producer)	? super T (Consumer)
Meaning	Some subtype of T	Some supertype of T
Safe For	**Reading** (get)	**Writing** (add)
Reading	Returns T	Returns Object
Writing	❌ Not allowed (except null)	✅ Can add T values
Example	List<? extends Number>	List<? super Integer>
Analogy	Basket producing Apples	Basket consuming Apples

Variance with wildcards & PECS:

? extends T → Producer Extends (you READ)

Think: “a basket that produces Ts for me to look at.”

```
double total(List<? extends Number> nums) {
    double s = 0;
    for (Number n : nums) s += n.doubleValue();
 // ✅ read as Number
    // nums.add(1);       // ❌ can't add (except null)
    return s;
}
```

- Why no adds? Because nums might actually be a List<Double> at runtime.
- If we allowed nums.add(1), we’d shove an Integer into a Double list.
- Rule: With ? extends T, you can read items as T, but you cannot add (except null).

? super T → Consumer Super (you WRITE)

Think: “a basket that can consume Ts I put into it.”

```
static <T> void copy(List<? super T> dst,
List<? extends T> src) {
    for (T t : src) {
        dst.add(t);  // ✅ safe to write T into a ? super
T
    }
    // T x = dst.get(0);
// ⚠️ reading gives only Object (type could be
supertype)
}
```

- If dst is List<Object> or List<Number>, it’s safe to add any T (like Integer) to it.
- Rule: With ? super T, you can write T into it safely, but reads are only guaranteed as Object.

- **invariance demo:**

List<Number> ln = new ArrayList<>();

List<Integer> li = new ArrayList<>();

// ln = li; // ❌ not allowed (invariance)

PECS : producer extends, consumers super

- If the parameter is a source you only read from → ? extends T
- If the parameter is a destination you write to → ? super T
- If you need to both read and write as the same T → use a type parameter <T> (no wildcard)

Situation	Use	Why
Summing numbers from a list	List<? extends Number>	You only read values as Number
Filling a list with integers	List<? super Integer>	You write Integers safely.
Copy from one list to another	copy(List<? super T> dst, List<? extends T> src)	Source produces (extends), dest consumes (super).
Sort with a comparator	Comparator<? super T>	A comparator of a **supertype** can compare Ts.
Transform with same in/out type	<T> T max(List<T>)	You both read and return T—use a type param, not wildcards.

Limitations of Generics

Name	Description	Snippet
No primitives as type arguments	You can't use int, double, etc. Use wrappers.	// ❌ Test<int> t = new Test<int>(); ✅ Test<Integer> t = new Test<>(); // Arrays are reference types, so this is fine: ✅ List<int[]> frames = new ArrayList<>();

No generic array creation	You can't create arrays of parameterized types or of T.	// ❌ List<String>[] a = new List<String>[10];
		// ❌ T[] arr = new T[10];
		✅ List<List<String>> a = new ArrayList<>(); // use lists of lists
		// (Raw arrays compiles with warnings–avoid.)

Can't use a class's type parameter in static context	static members don't see the instance type parameter.	class Repo<T> {
		// ❌ static T cache;
		static <U> Repo<U> empty() { return null; } //
		✅ declare your own <U>
		}

Wildcards are intentionally restrictive	? extends T is (effectively) read-only; ? super T is write-only (reads as Object).	void sum(List<? extends Number> xs) { /* can read Number, can't add */ } void addAll(List<? super Integer> dst) { dst.add(42); /* reads are Object */ }
No generic exceptions / catches	You can't make class MyEx<T> extends Exception {} and you can't catch (T e).	// ❌ class Bad<T> extends Exception {} // ❌ catch (T e) { ... }

Benefits of Generics

- Code Reusability
- Type Safety
- Individual type casting not needed

🎬 NETFLIX — “one feed, many item types”

Without Generics

```
// Non-generic page: items are just Objects (or FeedItem)
record PageRaw(List<Object> items, int page, int totalPages)
{}

class FeedServiceRaw {
  PageRaw loadRow(String rowId) {
    // could be movies or series... caller has to guess
    return new PageRaw(List.of(new Movie("Inception", 148)),
1, 1);
  }
}
```

How to call:

```
var page = new FeedServiceRaw().loadRow("continue");

// You must cast -- risky:
Movie m = (Movie) page.items().get(0);   // might throw
ClassCastException at runtime
```

- By polisetty Sai Teja

With Generics

```
interface FeedItem { String title(); }
record Movie(String title, int minutes) implements FeedItem {}
record Series(String title, int seasons) implements FeedItem {}

record Page<T>(List<T> items, int page, int totalPages) {} // T is the item
type

class FeedService {
  // T is any FeedItem subtype; Class<T> says WHICH subtype we want
  <T extends FeedItem> Page<T> loadRow(String rowId, Class<T> kind) {
    // imagine we fetched JSON and mapped each element to 'kind'
    List<T> items = fetchAndMap(rowId, kind);  // returns List<T>
    return new Page<>(items, 1, 10);
  }

  // stub to illustrate the idea
  private <T extends FeedItem> List<T> fetchAndMap(String rowId,
Class<T> kind) {
    // Example: if kind == Movie.class → build List<Movie>
    // If kind == Series.class → build List<Series>
    return List.of();
  }
}
```

How to call:

```
FeedService svc = new FeedService();

// Ask for movies:
Page<Movie> movies = svc.loadRow("continue", Movie.class);
Movie m = movies.items().get(0);     // ✅ already a Movie (no cast)

// Ask for series:
Page<Series> shows = svc.loadRow("trending", Series.class);
Series s = shows.items().get(0);     // ✅ already a Series
```

Few More Use cases

GOOGLE – “one search, many result kinds”

- “How can Google Search return a single results page that sometimes has Web, Image, Video results–each with different fields–without losing type safety?”

TESLA – “typed sensors & alerts”

- “How can telemetry code reuse the same pipeline for temperature, speed, voltage… but reject mixing types?”

Quiz

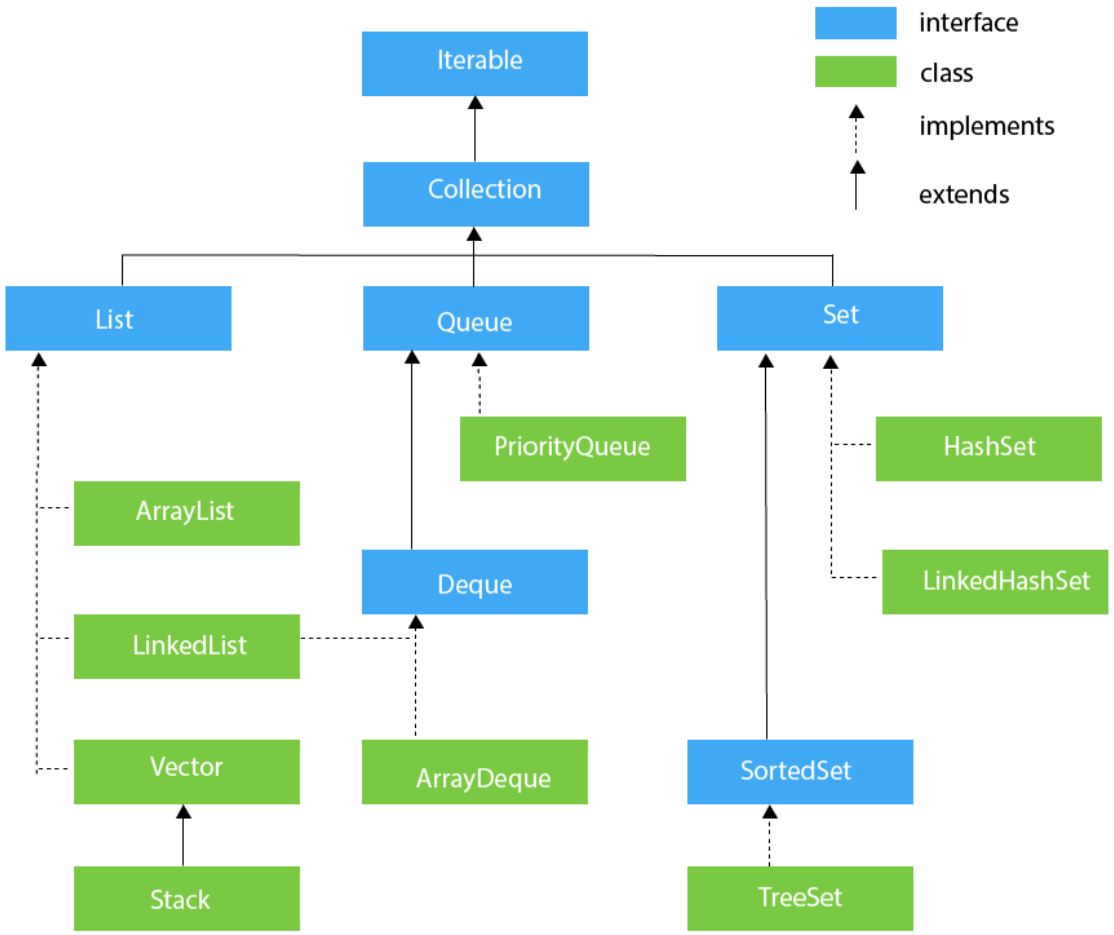
	Question	Options
1	Generics provide type _____.	A) safety B) casting C) reflection D) serialization
2	Java generics use type ____ at runtime.	A) erasure B) reification C) templates D) metadata
3	Default variance of List<T> is _____.	A) invariance B) covariance C) contravariance D) variance
4	In PECS, a **producer** uses _____.	A) extends B) super C) bounds D) wildcard
5	In PECS, a **consumer** uses _____.	A) super B) extends C) bounds D) wildcard
6	Wildcard symbol is _____.	A) ? B) T C) K D) V
7	Upper bound keyword is _____.	A) extends B) super C) implements D) bounds
8	Lower bound keyword is _____.	A) super B) extends C) below D) lower
9	A generic method declares type parameters before the ____ type.	A) return B) class C) package D) import
10	List<?> is effectively ____-only.	A) read B) write C) sync D) cast

	Question	Options
11	The wildcard rule acronym is _____.	A) PECS B) SOLID C) DRY D) MISS
12	Using a raw type (e.g., List) is generally _____.	A) unsafe B) faster C) immutable D) synchronized
13	Primitives as type arguments are _____.	A) disallowed B) promoted C) autoboxed D) nullable
14	new T() in a generic class is _____.	A) illegal B) generic C) cached D) reflective
15	Direct creation of generic arrays (T[]) is _____.	A) forbidden B) safe C) static D) trivial
16	Comparator<? ____ T> in sorting APIs typically uses _____.	A) super B) extends C) of D) for
17	Multiple bounds are joined with an _____.	A) ampersand B) comma C) plus D) pipe
18	The diamond operator enables type _____.	A) inference B) erasure C) boxing D) promotion
19	Type parameters cannot be referenced from a ____ context.	A) static B) dynamic C) generic D) override
20	To accept any subtype of Number,	A) extends B) super C) over D)

	Question	Answer
1	Generics provide type _____.	**safety** (A)
2	Java generics use type ____ at runtime.	**erasure** (A)
3	Default variance of List<T> is _____.	**invariance** (A)
4	In PECS, a **producer** uses _____.	**extends** (A)
5	In PECS, a **consumer** uses _____.	**super** (A)
6	Wildcard symbol is _____.	**?** (A)
7	Upper bound keyword is _____.	A) extends
8	Lower bound keyword is _____.	A) super
9	A generic method declares type parameters before the ____ type.	A) return
10	List<?> is effectively ____-only.	A) read

	Question	Answer
11	The wildcard rule acronym is _____.	A) PECS
12	Using a raw type (e.g., List) is generally _____.	A) unsafe
13	Primitives as type arguments are _____.	A) disallowed
14	new T() in a generic class is _____.	A) illegal
15	Direct creation of generic arrays (T[]) is _____.	A) forbidden
16	Comparator<? ____ T> in sorting APIs typically uses _____.	A) super
17	Multiple bounds are joined with an _____.	A) ampersand
18	The diamond operator enables type _____.	A) inference
19	Type parameters cannot be referenced from a ____ context.	A) static
20	To accept any subtype of Number, write ? ____ Number.	A) extends

Answer



List

- an ordered collection that allows duplicates. - Order matters (index based).
- “Think Spotify playlist–order matters, the same song can repeat.”
- Example:

```
List<String> names = new ArrayList<>();

names.add("A"); names.add("B");
names.add("A"); // duplicates allowed

System.out.println(names.get(1)); // "B"
```

Spotify — Playlist editor (ordering, duplicates, sorting, safe removal)

```
record Track(String id, String title, boolean explicit, int rating /*1..5*/) {}

class Playlist {
  private final List<Track> tracks = ??? ;

  void add(Track t) { ???  }                    // duplicates allowed
  void swap(int i, int j){ ???  }   // reorder
  void removeExplicit(){ ???  }   // safe bulk removal
  void sortByRating(){  ???  }

  // "Up next" window (first N) -- returns a *view* of the list
  List<Track> upNext(int n){
    return ???      // view, not a copy
  }
}
```

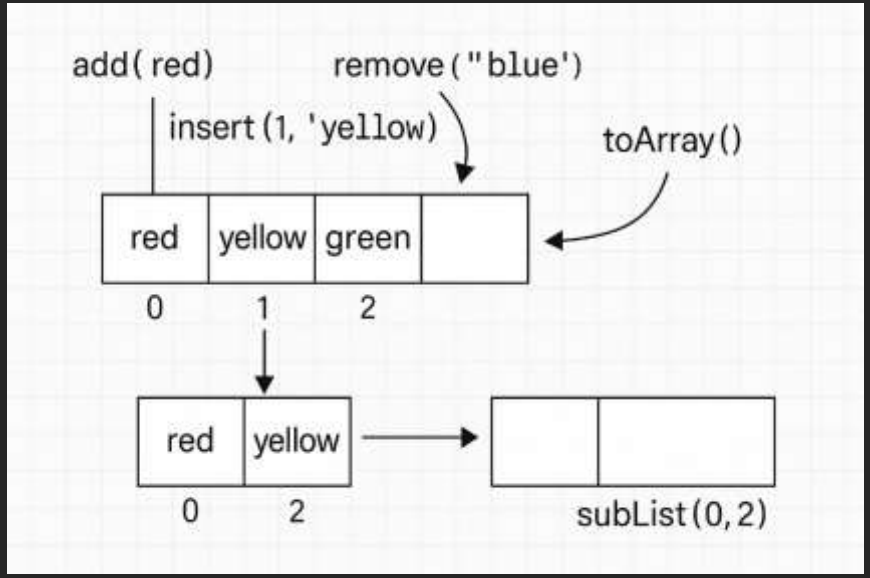
Implementation	Use when	Big O (Amortized)	Notes
ArrayList	general purpose; lots of reads; append-heavy	get O(1), add-end O(1), insert/remove middle O(n)	contiguous array; ensureCapacity helps; great default
LinkedList	mostly head/tail ops; lots of listIterator inserts/removes	get O(n), add/remove ends O(1)	higher memory; random access slow; usually prefer ArrayList or ArrayDeque for queues/stacks
CopyOnWriteArrayList	many reads, very few writes, iteration must never fail	get O(1), write O(n) (copies array)	snapshot iterators; great for listeners/observers
Collections.synchronizedList(...)	need a synchronized wrapper		must synchronized(list) { iterate } to iterate safely
Unmodifiable (List.of, List.copyOf, Collections.unmodifiableList)	you want read-only views or copies		of/copyOf throw on mutate & disallow nulls

Implementation

Quick rules:

- Default: ArrayList.
- For queue/stack: use ArrayDeque, not LinkedList.
- For read-mostly shared lists: CopyOnWriteArrayList.

Core Api's



- List<String> l = new ArrayList<>();
- l.add("red"); // append
- l.add(0, "blue"); // insert at index
- l.set(1, "green"); // replace
- l.remove("blue"); // remove by value (first match)
- l.remove(0); // remove by index
- boolean ok = l.contains("red");
- int i = l.indexOf("red"); // first index, -1 if not found
- List<String> view = l.subList(0, 1); // [0..1)
- l.sort(Comparator.naturalOrder()); // or l.sort(null) for natural
- l.replaceAll(String::toUpperCase);
- l.removeIf(s -> s.length() < 4);
- String[] arr = l.toArray(new String[0]); // or l.toArray(String[]::new) (Java 11+)

Immutability & fixed-size traps:

- var a = Arrays.asList("x","y"); // fixed-size, backed by array
- // a.add("z"); // UnsupportedOperationException
- a.set(0,"X"); // allowed (still fixed-size)

- var b = List.of("x","y"); // truly unmodifiable (Java 9+)
- // b.set(0,"X"); // UnsupportedOperationException
- // b.add("z"); // UnsupportedOperationException

- var c = Collections.unmodifiableList(new ArrayList<>(a)); // read-only view of a *copy*

- “Arrays.asList is a hotel room with nailed-down furniture;
- List.of is the museum—no touching.”

SubList is a view

- subList(from, to) returns a window into the original list.
- Structural changes to parent or sublist can cause ConcurrentModificationException.
- If you need independence: new ArrayList<>(list.subList(...))

```
var base = new ArrayList<>(List.of(10,20,30,40,50));
var mid = base.subList(1,4);      // [20,30,40]
mid.set(0, 200); // base now [10,200,30,40,50]
var copy = new ArrayList<>(mid);
```

Iteration & fail-fast vs safe & sorting

- Fail fast iterator:

```
for (String s : list) {
  if (s.startsWith("X")) {
    // list.remove(s); // ✕
ConcurrentModificationException
  }
}
// Correct:
Iterator<String> it = list.iterator();
while (it.hasNext()) {
  if (it.next().startsWith("X")) it.remove(); // ✓
}
```

- Safe iteration under concurrency

CopyOnWriteArrayList: iteration sees a snapshot; writes copy the array (costly for frequent writes).

Collections.synchronizedList: wrap + synchronized(list) { for(...) } around iteration.

- ListIterator for in-place edits + backwards

```
ListIterator<String> it2 = list.listIterator();
while (it2.hasNext()) {
  String s = it2.next();
  if (s.equals("A")) it2.set("Alpha");
  if (s.equals("B")) it2.add("Beta"); // inserts before next()
}
while (it2.hasPrevious()) {
  System.out.println(it2.previous());
}
```

- Sorting & comparators:

list.sort(null) → natural order (elements must be Comparable).

Custom comparator:

```
list.sort(Comparator.comparing(User::score).reversed()
                .thenComparing(User::name));
```

Why Comparator<? super T>? A comparator of a supertype can compare subtypes—variance for flexibility.

Performance:

ArrayList appends are amortized O(1); middle inserts/removes O(n) (shift).

ensureCapacity(n) before big appends avoids repeated grows.

trimToSize() can free memory after big removals.

LinkedList: each node has pointers → higher memory, slow random access; shines only for frequent head/tail ops with iterator already positioned.

Need a queue/stack? Prefer ArrayDeque (faster, no capacity boxing).

Spotify — Playlist editor (ordering, duplicates, sorting, safe removal)

```
record Track(String id, String title, boolean explicit, int rating /*1..5*/)
{}


class Playlist {
  private final List<Track> tracks = new ArrayList<>();


  void add(Track t) { tracks.add(t); }                    // duplicates
allowed
  void swap(int i, int j){ Collections.swap(tracks, i, j); }   // reorder
  void removeExplicit(){ tracks.removeIf(Track::explicit); }   // safe bulk
removal
  void sortByRating(){
tracks.sort(Comparator.comparingInt(Track::rating).reversed()); }


  // "Up next" window (first N) -- returns a *view* of the list
  List<Track> upNext(int n){
    return tracks.subList(0, Math.min(n, tracks.size()));      // view, not a
copy
  }
}
```

- **Points to remember:**
  - Lists are ordered and allow duplicates → perfect for playlists.
  - Use removeIf instead of removing inside a for-each (avoids CME).
  - subList is a view (changes reflect back).

Example Use cases:

Tesla – Sensor sliding window (last N / last via subList + binary search)

Charts of the last 60 seconds of telemetry.

List idea: Keep readings time-sorted in a List<Reading>; cut a sliding window with subList.

Netflix – “Continue Watching” row (recency bump, pagination via subList, iterator edits)

Recently watched titles show first; pagination shows only first 10.

List idea: Keep a recency-ordered List<Title> and bump to front on activity.

Quiz

	Question	Options
1	A List preserves _____.	A) order B) hashing C) uniqueness D) immutability
2	A List allows _____.	A) duplicates B) nulls C) keys D) sorting
3	Default go-to implementation?	A) ArrayList B) LinkedList C) Vector D) Stack
4	Random access time in ArrayList is _____.	A) constant B) linear C) logarithmic D) quadratic
5	Random access time in LinkedList is _____.	A) linear B) constant C) logarithmic D) quadratic
6	Preferred structure for queue/stack (not LinkedList) is _____.	A) ArrayDeque B) Vector C) ArrayList D) Deque
7	Removing while for-each iterating causes _____.	A) CME B) NPE C) OOM D) IAE
8	Safe bulk removal method is _____.	A) removeIf B) clear C) delete D) prune
9	remove(2) on List<Integer> removes by _____.	A) index B) value C) key D) hash
10	To remove the integer value 2, call _____.	A) valueOf B) parseInt C) equals D) compare

	Question	Options
11	subList returns a _____.	A) view B) copy C) clone D) cache
12	Arrays.asList is ____-size.	A) fixed B) variable C) dynamic D) infinite
13	List.of is _____.	A) unmodifiable B) synchronized C) nullable D) growable
14	Bulk iteration that won't CME under reads-mostly workload:	A) CopyOnWriteArrayList B) LinkedList C) Vector D) Stack
15	Synchronized wrapper factory is _____.	A) synchronizedList B) syncList C) lockList D) guardedList
16	To iterate a synchronized list safely, you must _____.	A) synchronize B) finalize C) serialize D) localize
17	Sorting natural order uses _____.	A) sort B) order C) rank D) align
18	Equality in indexOf relies on _____.	A) equals B) hashCode C) identity D) compareTo
19	Marker interface for fast random access is _____.	A) RandomAccess B) FastAccess C) QuickIndex D) DirectIndex
20	Stream.toList() (Java 16+) returns a _____ list.	A) unmodifiable B) synchronized C) mutable D) growable

	Question	Answers
1	A List preserves _____.	A) order
2	A List allows _____.	A) duplicates
3	Default go-to implementation?	A) ArrayList
4	Random access time in ArrayList is _____.	A) constant
5	Random access time in LinkedList is _____.	A) linear
6	Preferred structure for queue/stack (not LinkedList) is _____.	A) ArrayDeque
7	Removing while for-each iterating causes _____.	A) CME
8	Safe bulk removal method is _____.	A) removeIf
9	remove(2) on List<Integer> removes by _____.	A) index
10	To remove the integer value 2, call _____.	A) valueOf

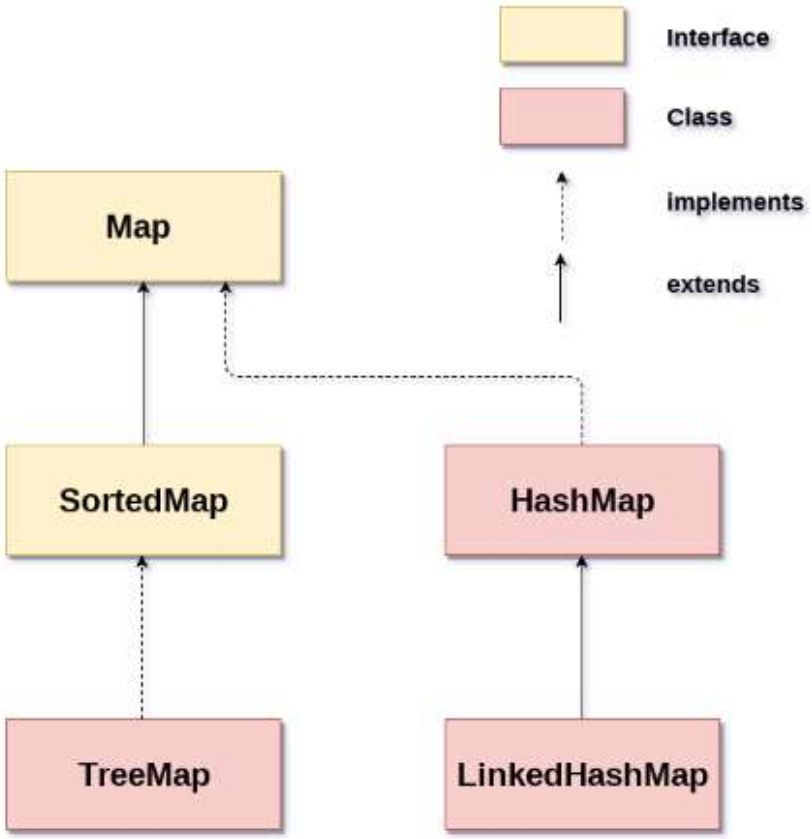
	Question	Options
11	subList returns a _____.	A) view
12	Arrays.asList is ____-size.	A) fixed
13	List.of is _____.	A) unmodifiable
14	Bulk iteration that won't CME under reads-mostly workload:	A) CopyOnWriteArrayList
15	Synchronized wrapper factory is _____.	A) synchronizedList
16	To iterate a synchronized list safely, you must _____.	A) synchronize
17	Sorting natural order uses _____.	A) sort
18	Equality in indexOf relies on _____.	A) equals
19	Marker interface for fast random access is _____.	A) RandomAccess
20	Stream.toList() (Java 16+) returns a ____ list.	A) unmodifiable

Answers

Map

- A Map<K,V> stores key → value pairs with unique keys.
  - Lookup by key (not by index).
  - Typical ops: put, get, containsKey, remove, size, isEmpty.
  - Views: keySet(), values(), entrySet()

```
Map<String, String> phone = new HashMap<>();
phone.put("Alice", "999-111");
phone.put("Bob",   "999-222");
System.out.println(phone.get("Alice"));     // 999-111
System.out.println(phone.containsKey("Eve"));// false
```



Uber/Ola — live drivers by area (computeIfAbsent + removeIf) Match riders fast by grid cell.

```
record Driver(String id) {}
record Cell(int x, int y) {}
Map<Cell, List<Driver>> available = ???
;

void appear(Cell c, Driver d) {
  ???
}

void acceptRide(Driver d) { ??? }
```

Implementation	Use when	Ordering	Nulls	Time (avg)
HashMap	general purpose; fastest lookups	none	allows 1 null key, many null values	get/put ~ O(1)
LinkedHashMap	you need insertion order (or LRU)	insertion or access	allows nulls	O(1)
TreeMap	you need sorted keys / range queries	sorted (Comparator)	no null keys (NPE)	O(log n)
ConcurrentHashMap	multi-threaded reads/writes without external locks	none	no nulls	O(1) expected
EnumMap	keys are a single enum type	enum order	no null keys	O(1)
WeakHashMap	auto-remove entries when keys GC'd (cache keys)	none	allows nulls	O(1)
IdentityHashMap	key identity == (not equals)	none	allows nulls	O(1)

Implementation

- Quick rules:
  - Default: HashMap
  - Need order or LRU cache: LinkedHashMap
  - Need range queries (e.g., "next higher key"): TreeMap
    Threads: ConcurrentHashMap (CHM)

Core Api's:

```
Map<Integer,String> m = new HashMap<>();
m.put(101, "Alice");
m.putIfAbsent(101, "New");        // won't overwrite
m.putIfAbsent(102, "Bob");

String name = m.getOrDefault(999, "Unknown");

m.replace(102, "Bobby");          // replace if present
m.computeIfAbsent(103, k -> "Temp"); // lazy create
m.merge(103, "X", (oldV, v) -> oldV + v); // combine

// Iteration patterns
for (Map.Entry<Integer,String> e : m.entrySet()) {
  System.out.println(e.getKey() + " → " + e.getValue());
}
m.forEach((k,v) -> System.out.println(k + ":" + v));
```

Merge (word frequency):

```
Map<String,Integer> freq = new HashMap<>();
for (String w : words) freq.merge(w, 1, Integer::sum);
```

Ordering tricks:

- **LinkedHashMap (LRU cache in 6 lines):**

```
class LruCache<K,V> extends LinkedHashMap<K,V> {
  private final int cap;
  LruCache(int cap) { super(16, 0.75f, true); this.cap = cap; } // access-order
  protected boolean removeEldestEntry(Map.Entry<K,V> e) { return size() > cap; }
}
```

- **TreeMap (sorted + range queries):**

```
TreeMap<Integer,String> tm = new TreeMap<>();
tm.put(10,"A"); tm.put(20,"B"); tm.put(30,"C");
tm.floorKey(21);   // 20
tm.ceilingEntry(19).getValue(); // "B"
tm.subMap(10, true, 20, false); // [10..20)
```

Equality, hashing & the #1 Map bug:

Maps rely on key equality:

HashMap uses hashCode() to choose a bucket, then equals() to confirm.
Contract: if a.equals(b) then a.hashCode()==b.hashCode().

Don’t mutate a key after put! If fields used in equals/hashCode change, the entry becomes “lost” in the wrong bucket.

```
record UserId(String tenant, String id) { } // records give
    correct equals+hashCode

Map<UserId, Profile> profiles = new HashMap<>();

UserId k = new UserId("t1", "42");

profiles.put(k, new Profile());

// k.tenant = "t2"; // (if mutable) retrieval breaks!
```

- Null rules (surprises):

Type	Rule
HashMap/LinkedHashMap	allow one null key + many null values.
TreeMap	no null key with natural ordering (NPE).
ConcurrentHashMap	no null keys or values (to avoid ambiguity with “missing”).

Safe removal while iterating:

```
// ✅ safest: operate on view collections
m.entrySet().removeIf(e -> e.getValue().isBlank());

// or explicit iterator
for (Iterator<Map.Entry<K,V>> it = m.entrySet().iterator();
it.hasNext();) {
  var e = it.next();
  if (shouldRemove(e)) it.remove();
}
```

Concurrency quick guide:

To iterate safely:

HashMap + sync: Map<K,V> sm = Collections.synchronizedMap(new HashMap<>());

synchronized (sm) { for (var e : sm.entrySet()) {/*...*/} }

ConcurrentHashMap:

No nulls; iterators don’t throw CME.

Methods like compute/merge/putIfAbsent are atomic per key.

Immutability & factory methods:

- ** Map.of rejects null keys/values and throws on modification.

```
Map<String,Integer> m1 = Map.of("a",1,"b",2);    // unmodifiable
Map<String,Integer> m2 = Map.copyOf(m1);        // defensive copy (alsounmodifiable)
Map<String,Integer> ro = Collections.unmodifiableMap(newHashMap<>(m1)); // view of a copy
```

Uber/Ola — live drivers by area (computeIfAbsent + removeIf) Match riders fast by grid cell.

```
record Driver(String id) {}
record Cell(int x, int y) {}
Map<Cell, List<Driver>> available = new
HashMap<>();

void appear(Cell c, Driver d) {
  available.computeIfAbsent(c, k -> new
ArrayList<>()).add(d);
}
void acceptRide(Driver d) {
available.values().forEach(list -> list.removeIf(x ->
x.id().equals(d.id())));
}
```

**Why Map? Directly hits the right bucket (cell) for O(1)-ish candidate retrieval.

**Slight improvement with concurrenthashmap.

Example use cases:

Netflix – “continue watching” progress (merge)	Count minutes watched per title per profile.
Spotify – user → playlists (computeIfAbsent)	Each user has many playlists; each playlist has tracks.
Tesla – time-series lookups (TreeMap range/floor)	Find the nearest reading at/after a timestamp.
Instagram/Twitter – trending hashtags (ConcurrentHashMap + merge)	Many threads updating counts safely.

Quiz

	Question	Options
1	General-purpose Map with O(1) lookups?	A) HashMap B) TreeMap C) LinkedHashMap D) EnumMap
2	Map preserving insertion order?	A) TreeMap B) LinkedHashMap C) HashMap D) WeakHashMap
3	Map with keys kept sorted?	A) TreeMap B) HashMap C) LinkedHashMap D) IdentityHashMap
4	High-throughput thread-safe Map?	A) Hashtable B) ConcurrentHashMap C) WeakHashMap D) EnumMap
5	Map specialized for enum keys?	A) TreeMap B) EnumMap C) HashMap D) IdentityHashMap
6	Map comparing keys by reference identity?	A) IdentityHashMap B) HashMap C) TreeMap D) EnumMap
7	Map that drops entries when keys are GC'd?	A) WeakHashMap B) ConcurrentHashMap C) LinkedHashMap D) TreeMap
8	With new LinkedHashMap(16,0.75f, true), ordering is _____.	A) insertion B) access C) hash D) random
9	Create value container if missing: _____.	A) merge B) computeIfAbsent C) putIfAbsent D) getOrDefault
10	Combine counts (old + 1): _____.	A) replace B) merge C) compute D) put

	Question	Options
11	Return fallback when key missing: _____.	A) getOrDefault B) orElse C) optional D) defaultGet
12	Remove only if key maps to given value: _____.	A) remove B) delete C) clear D) prune
13	Unmodifiable factory with literals: _____.	A) Map.of B) Map.copyOf C) Collections.emptyMap D) Map.new
14	Unmodifiable defensive copy of another Map: _____.	A) Map.copyOf B) Map.of C) unmodifiableMap D) clone
15	Nearest key ≤ k: _____.	A) lowerKey B) ceilingKey C) floorKey D) higherKey
16	Nearest entry ≥ k: _____.	A) ceilingEntry B) floorEntry C) higherEntry D) lowerEntry
17	Range view between two keys: _____.	A) headMap B) tailMap C) subMap D) sliceMap
18	View for iterating key+value pairs: _____.	A) keySet B) values C) entrySet D) pairSet
19	Null keys in HashMap are allowed: _____.	A) one B) many C) none D) some
20	Null keys in ConcurrentHashMap are	A) one B) many C) none D) some

Answers

	Question	Answers
1	General-purpose Map with O(1) lookups?	A) HashMap
2	Map preserving insertion order?	A) TreeMap
3	Map with keys kept sorted?	A) TreeMap
4	High-throughput thread-safe Map?	A) Hashtable
5	Map specialized for enum keys?	A) TreeMap
6	Map comparing keys by reference identity?	A) IdentityHashMap
7	Map that drops entries when keys are GC'd?	A) WeakHashMap
8	With new LinkedHashMap(16,0.75f, true), ordering is _____.	A) insertion
9	Create value container if missing: _____.	A) merge
10	Combine counts (old + 1): _____.	A) replace

	Question	Answers
11	Return fallback when key missing: _____.	A) getOrDefault
12	Remove only if key maps to given value: _____.	A) remove
13	Unmodifiable factory with literals: _____.	A) Map.of
14	Unmodifiable defensive copy of another Map: _____.	A) Map.copyOf
15	Nearest key ≤ k: _____.	A) lowerKey
16	Nearest entry ≥ k: _____.	A) ceilingEntry
17	Range view between two keys: _____.	A) headMap
18	View for iterating key+value pairs: _____.	A) keySet
19	Null keys in HashMap are allowed: _____.	A) one
20	Null keys in ConcurrentHashMap are allowed: _____.	A) one