Module 13 of 20

Generics

Why generics exist, generic classes and methods, bounded types, wildcards and PECS, type erasure and its consequences, and the practical idioms.

Generics are how Java's type system describes collections, functions and containers precisely: a list of strings, a function from T to R, a repository of entities. They were bolted onto a running language in 2004 through erasure, and that history explains both their power and every one of their sharp edges.

This module builds the model from the ground up: the problem generics solve and the vocabulary; writing generic classes, interfaces and methods and how inference works; bounded type parameters including the self-referential Comparable bound; wildcards, invariance and the PECS rule for flexible signatures; erasure and the full list of things it makes impossible, with bridge methods and heap pollution; and the everyday idioms and compiler messages.

By the end you read any JDK signature, write generic utilities with correct bounds and wildcards, and explain erasure and its consequences in an interview.

CONTENTS

1	Why generics exist	12 min
2	Writing generic classes and methods	14 min
3	Bounded type parameters	12 min
4	Wildcards and PECS	15 min
5	Type erasure and its consequences	14 min
6	Generics in practice — idioms and error messages	12 min
✓	Checkpoint — Generics	24 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

Why generics exist

Before Java 5, a `List` held `Object`s: you put a `String` in and got an `Object` out, cast it back, and found out at run time if you were wrong. Generics let you say `List<String>` — a list *of strings* — so the compiler checks what goes in and knows what comes out. This lesson is the problem generics solve, the vocabulary (type parameter, type argument, parameterised type, raw type), and the first look at the design decision — erasure — that explains every limitation you will meet later in the module.

The problem

```
List names = new ArrayList();            // pre-generics: a list of Object
names.add("Ada");
names.add(42);                           // nothing stops this
String first = (String) names.get(0);    // cast required…
String second = (String) names.get(1);   // …and this one throws ClassCastExce
```

Three costs: casts everywhere, no documentation of what the list holds, and errors that surface far from their cause — the `42` was added in one file and exploded in another.

The solution

```
List<String> names = new ArrayList<>();
names.add("Ada");
names.add(42);                           // compile error: incompatible types:
String first = names.get(0);             // no cast: the compiler knows it is
```

`List<String>` is a **parameterised type**: the generic interface `List<E>` with the **type argument** `String` supplied for its **type parameter** `E`. Inside `List`, every `E` now means `String`, so `add(E)` accepts only strings and `get(int)` returns one. The error moves to compile time, to the line that is actually wrong.

Vocabulary

Term	Example	Meaning
Generic type	`List<E>`, `Map<K, V>`	A type declared with type parameters
Type parameter	`E`, `K`, `V`, `T`	A placeholder in the declaration
Type argument	`String` in `List<String>`	The actual type supplied at use
Parameterised type	`List<String>`	A generic type with arguments
Raw type	`List`	A generic type used without arguments — legacy, unsafe
Generic method	`static <T> T first(List<T> xs)`	A method with its own type parameters
Bounded type	`<T extends Comparable<T>>`	A parameter restricted to subtypes of something
Wildcard	`List<?>`, `List<? extends Number>`	An unknown type argument, possibly bounded

Conventions: single capital letters — `T` (type), `E` (element), `K` / `V` (key/value), `R` (result), `N` (number). Two-letter names are unusual; descriptive names are not idiomatic.

Type arguments must be reference types

`List<int>` is a compile error; `List<Integer>` with autoboxing is how primitives get in. This costs boxing (Module 2) and is the reason `IntStream`, `int[]` and primitive-specialised functional interfaces exist. Project Valhalla is working on lifting this; for now, wrappers.

The diamond

```
Map<String, List<Integer>> index = new HashMap<String, List<Integer>>();   //
Map<String, List<Integer>> index = new HashMap<>();                         //
var index = new HashMap<String, List<Integer>>();                           //
```

`<>` tells the compiler to infer the constructor's type arguments from the target type. Combine it with `var` and the arguments must be on the right instead — never `var x = new HashMap<>()`, which infers `HashMap<Object, Object>`.

Raw types: why they still exist and why you avoid them

`List` without arguments is a **raw type**. It compiles for backwards compatibility with pre-2004 code, with warnings ("unchecked call"), and it turns off checking:

```
List<String> safe = new ArrayList<>();
List raw = safe;                          // allowed, warning
raw.add(42);                              // allowed, warning — pollutes the St
String s = safe.get(0);                   // ClassCastException here, far from
```

Never write a raw type in new code. The one legitimate appearance is in class literals and `instanceof` (`List.class`, `x instanceof List`), because those cannot carry type arguments — the next section explains why.

Erasure, in one paragraph

Generics were added to a language with ten years of compiled libraries and a JVM that knew nothing about them. The design chosen was **erasure**: type arguments exist for the compiler only. After checking, `List<String>` compiles to plain `List`, `T` becomes `Object` (or its bound), and casts are inserted where the code reads elements. At run time there is one `List` class, no `List<String>`, and `new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass()` is `true`. This kept old code running unchanged and is the reason for every "you cannot do that with generics" rule in lesson 5. Understand it now and the rules will feel inevitable rather than arbitrary.

What you gain, concretely

Compile-time safety: wrong-type insertions are errors at the line that is wrong.

No casts in reading code.

Documentation: `Map<CustomerId, List<Order>>` says what a structure holds; `Map` says nothing.

Reusable algorithms: one `sort`, one `max`, one `filter` for every element type, without `Object` and casts.

APIs that describe themselves: `Function<String, Integer>` tells you the whole contract.

Reading generic signatures

`public static <T extends Comparable<? super T>> void sort(List<T> list)` — read it left to right: a generic method with parameter `T`, where `T` must be comparable to itself or a supertype; takes a list of `T`; returns nothing. By the end of this module every part of that line will be obvious. Until then, the skill is to not panic: find the type parameters, their bounds, and where they appear in the parameters and return.

Interview angle

"What are generics for?" Compile-time type safety and reuse without casts; errors move to the source line.

"What is a raw type and why avoid it?" A generic type without arguments; it disables checking and pollutes typed collections.

"Can you have a `List<int>`?" No — type arguments must be reference types; use `Integer` (boxing) or `int[]`.

"What is erasure?" Type arguments are compile-time only; at run time `List<String>` is `List`.

Key takeaways

Generics move type errors from run time (casts) to compile time (the wrong line) and document what collections hold.

Vocabulary: type parameter `E`, type argument `String`, parameterised type `List<String>`, raw type `List`.

Reference types only; the diamond `<>` infers constructor arguments from the target.

Raw types are legacy — never in new code except class literals and `instanceof`.

Erasure: generics are for the compiler; run time sees the raw class.

EXERCISE 1

Typed from the start

Read an integer `n` and `n` tokens. Put the integers into a `List<Integer>` and everything else into a `List<String>` (a token is an integer if `matches("-?\\d+")`). Print `ints=<sum of the integers>` and `words=<the strings joined by ,>` (or `words=` when there are none), then a third line `same class=<whether the two lists' getClass() are ⟹` to observe erasure.

Example: `4` then `3 a 4 b` →

```
ints=7
words=a,b
same class=true
```

INPUT 1

```
4
3 a 4 b
```

OUTPUT 1

```
ints=7
words=a,b
same class=true
```

INPUT 2

```
2
-1 -2
```

OUTPUT 2

```
ints=-3
words=
same class=true
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> ints = new ArrayList<>();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String t = in.next();
            if (t.matches("-?\\d+")) ints.add(Integer.parseInt(t));
            else words.add(t);
        }
        long sum = 0;
        for (int v : ints) sum += v;
        System.out.println("ints=" + sum);
        System.out.println("words=" + String.join(",", words));
        System.out.println("same class=" + (ints.getClass() == words.getClass()));
    }
}
```

EXERCISE 2

A typed pair

Write `class Pair<A, B>` with `first()`, `second()`, `Pair<B, A> swap()` and a `toString` of `(first, second)`. Read `n` lines `<word> <int>`; build a `Pair<String, Integer>` for each, print it, print its `swap()`, and add `second()` to a running total **without a cast** — the type parameter makes `second()` an `Integer`. Finally print `total=<sum>`.

Example: `2` then `ab 3`, `cd 4` →

```
(ab, 3) swapped (3, ab)
(cd, 4) swapped (4, cd)
total=7
```

INPUT 1

```
2
ab 3
cd 4
```

OUTPUT 1

```
(ab, 3) swapped (3, ab)
(cd, 4) swapped (4, cd)
total=7
```

INPUT 2

```
1
z -1
```

OUTPUT 2

```
(z, -1) swapped (-1, z)
total=-1
```

REFERENCE SOLUTION

```
import java.util.*;

class Pair<A, B> {
    private final A first;
    private final B second;
    Pair(A first, B second) { this.first = first; this.second = second; }
    A first() { return first; }
    B second() { return second; }
    Pair<B, A> swap() { return new Pair<>(second, first); }
    @Override public String toString() { return "(" + first + ", " + second + ")"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int total = 0;
        for (int i = 0; i < n; i++) {
            Pair<String, Integer> p = new Pair<>(in.next(), in.nextInt());
            System.out.println(p + " swapped " + p.swap());
            total += p.second();
        }
        System.out.println("total=" + total);
    }
}
```

QUIZ

1. In `List<String>`, `String` is the...

A. Type parameter

B. Type argument

C. Raw type

D. Bound

2. `List<int>` fails because…

A. `int` is reserved

B. Type arguments must be reference types; use `Integer`

C. Lists cannot hold numbers

D. It needs `<Int>`

3. Using a raw type such as `List` in new code…

A. Is recommended for speed

B. Disables type checking and can pollute typed collections

C. Is a compile error

D. Makes the list immutable

4. `new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass()` is…

A. `false`

B. `true` — erasure leaves one run-time class

C. A compile error

D. Undefined

5. `var m = new HashMap<>();` infers…

A. `HashMap<String, Object>`

B. `HashMap<Object, Object>` — there is no target type to infer from

C. A compile error

D. `HashMap<?, ?>`

ANSWER KEY · WHY GENERICS EXIST

1. B `E` in `List<E>` is the parameter; `String` is the argument supplied for it; `List<String>` is the parameterised type.

2. B Erasure turns type parameters into `Object`, which cannot hold a primitive. Autoboxing bridges the gap.

3. B Raw types exist for pre-generics code; the compiler warns because they let wrong-typed elements in.

4. B Type arguments are compile-time only. There is no `ArrayList<String>` class at run time.

5. B With `var`, put the type arguments on the right-hand side instead of the diamond.

LESSON 2 · 14 MIN

Writing generic classes and methods

Using `List<String>` is the easy half. Writing your own generic types — a `Pair<A, B>`, a `Stack<T>`, a `Result<T>`, a `static <T> T firstOrDefault(List<T>, T)` — is where generics stop being syntax and become a design tool. This lesson covers declaring type parameters on classes, interfaces and methods, how the compiler infers type arguments, and the common mistakes when a class and a method both introduce `T`.

A generic class

```
public final class Pair<A, B> {
    private final A first;
    private final B second;

    public Pair(A first, B second) { this.first = first; this.second = second;
    public A first()  { return first; }
    public B second() { return second; }
    public <C> Pair<A, C> withSecond(C c) { return new Pair<>(first, c); }

    @Override public String toString() { return "(" + first + ", " + second +
}

Pair<String, Integer> p = new Pair<>("age", 30);
Pair<String, Double> q = p.withSecond(30.5);
```

The type parameters go after the class name and are in scope throughout the body: fields, constructor parameters, method signatures, nested non-static types. Inside `Pair<A, B>`, `A` and `B` are just types you cannot instantiate or inspect (erasure — lesson 5), but you can store, pass and return them. (A record `record Pair<A, B>(A first, B second) {}` does all of this in one line — records are generic too.)

A generic stack

```
public class Stack<T> {
    private Object[] items = new Object[8];        // NOT new T[8] — impossibl
    private int size;

    public void push(T item) {
        if (size == items.length) items = Arrays.copyOf(items, size * 2);
        items[size++] = item;
    }

    @SuppressWarnings("unchecked")
    public T pop() {
        if (size == 0) throw new NoSuchElementException();
        T item = (T) items[--size];                 // the one unavoidable unc
        items[size] = null;                         // let the GC have it
        return item;
    }

    public boolean isEmpty() { return size == 0; }
}
```

The backing store is `Object[]` and the cast on the way out is unchecked; the class is still safe, because only `push(T)` ever stores into the array. `@SuppressWarnings("unchecked")` on the smallest possible scope, with the reason clear from the code, is the idiom. This is exactly how `ArrayList` is written.

Generic interfaces

```
public interface Repository<T, ID> {
    Optional<T> findById(ID id);
    List<T> findAll();
    T save(T entity);
}
public class UserRepository implements Repository<User, Long> { … }     // fix
public class InMemoryRepository<T, ID> implements Repository<T, ID> { … }   //
```

`Comparable<T>`, `Iterable<T>`, `Function<T, R>` and every collection interface follow this shape. When you implement one, either supply concrete arguments or carry your own parameters through.

Generic methods

```
public static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}
public static <T extends Comparable<T>> T max(List<T> list) { … }          //
public static <K, V> Map<V, K> invert(Map<K, V> map) { … }
public static <T> void swap(T[] array, int i, int j) { … }

String s = firstOrDefault(names, "none");          // T inferred as String fr
Integer n = Main.<Integer>firstOrDefault(List.of(), 0);   // explicit type wi
```

The type parameter list goes **before the return type**. A generic method's parameters are inferred per call from the argument types and the target type, so callers almost never write them. Generic methods may be static — and must be, if they are utilities on a non-generic class; a *class* type parameter cannot be used in a static method (there is no instance to fix it), which is why `static <T>` declares its own.

Inference

The compiler infers type arguments from: the argument expressions, the target type (assignment, return, parameter), and bounds. It is good but not magic:

```
List<String> empty = Collections.emptyList();       // T = String from the ta
var e = Collections.emptyList();                    // T = Object — no target
process(Collections.emptyList());                   // Java 8+ infers from pr
List<Number> nums = List.of(1, 2.0);                // T inferred as Number &
```

When inference fails ("incompatible types: inference variable T has incompatible bounds"), add a witness (`Collections.<String>emptyList()`), a typed local variable, or a lambda parameter type.

Class parameter versus method parameter

```
public class Box<T> {
    private T value;
    public <T> void confuse(T other) { … }        // a NEW, unrelated T that S
    public void set(T v) { value = v; }           // the class's T
    public <U> Box<U> map(Function<T, U> f) { return new Box<>(f.apply(value)
}
```

Redeclaring the class's parameter name on a method creates a second, independent parameter and silently breaks the connection. Use a fresh letter for method-level parameters inside generic classes.

Static members and type parameters

```
public class Registry<T> {
    private static T shared;                  // compile error: non-static type v
    private static int count;                 // fine
    public static <T> Registry<T> empty() { return new Registry<>(); }   // it
}
```

The class's `T` is per-instance in concept; statics are per-class, and there is one class for all `T`s under erasure. A static factory declares its own type parameter.

Multiple parameters and naming

`Map<K, V>`, `BiFunction<T, U, R>`, `Pair<A, B>`: as many as needed, comma-separated. Keep to single capitals; name by role (`K`, `V`, `E`, `R`); do not reuse a name in nested scopes.

Generic constructors and nested types

Constructors may declare their own type parameters (`<T> Box(T seed)`) — rare. A non-static inner class of a generic class sees the outer's parameters (`Box<T>.Inner` may use `T`); a static nested class does not (`static class Node<T>` needs its own).

Interview angle

"Where does a generic method's type parameter go?" Before the return type: `static <T> T f(…)`.

"Can a static method use the class's type parameter?" No — it declares its own.

"How do you create a `T[]` inside a generic class?" You cannot directly; use `Object[]` with a contained unchecked cast, or `Array.newInstance` with a `Class<T>`.

"What is the diamond?" `<>` — infer the constructor's type arguments from the target type.

"What goes wrong if a method redeclares `<T>` inside `Box<T>`?" It shadows the class parameter with an unrelated one.

Key takeaways

Class/interface parameters go after the name and are in scope for the body; method parameters go before the return type and are inferred per call.

Backing arrays are `Object[]` with one contained, suppressed cast — the `ArrayList` pattern.

Statics cannot use the class's parameter; declare their own. Never shadow a class parameter on a method.

Inference uses arguments, target type and bounds; give a witness or a typed variable when it fails.

EXERCISE 1

A generic stack and a generic helper

Write `class Stack<T>` backed by an `Object[]` with `push`, `pop`, `peek` and `isEmpty` (one contained `@SuppressWarnings("unchecked")` cast). Write `static <T> String describe(Stack<T> s, int take)` that pops up to `take` elements and returns them joined by `,`.

Read an integer `n` and `n` tokens; push them onto a `Stack<String>`; then read `k` and print `describe(stack, k)`. Also push the integers 1..n onto a `Stack<Integer>` and print `describe(ints, k)` — the same generic method serving both.

Example: `3` then `a b c`, `k = 2` →

```
c,b
3,2
```

INPUT 1

```
3
a b c
2
```

OUTPUT 1

```
c,b
3,2
```

INPUT 2

```
2
x y
5
```

OUTPUT 2

```
y,x
2,1
```

REFERENCE SOLUTION

```
import java.util.*;

class Stack<T> {
    private Object[] items = new Object[4];
    private int size = 0;

    void push(T item) {
        if (size == items.length) items = Arrays.copyOf(items, size * 2);
        items[size++] = item;
    }

    @SuppressWarnings("unchecked")
    T pop() {
        if (size == 0) throw new NoSuchElementException();
        T item = (T) items[--size];     // safe: only push(T) stores here
        items[size] = null;
        return item;
    }

    @SuppressWarnings("unchecked")
    T peek() {
        if (size == 0) throw new NoSuchElementException();
        return (T) items[size - 1];
    }

    boolean isEmpty() { return size == 0; }
}

public class Main {
    static <T> String describe(Stack<T> s, int take) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < take && !s.isEmpty(); i++) {
            if (sb.length() > 0) sb.append(',');
            sb.append(s.pop());
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Stack<String> words = new Stack<>();
        Stack<Integer> ints = new Stack<>();
        for (int i = 0; i < n; i++) words.push(in.next());
        for (int i = 1; i ⩽ n; i++) ints.push(i);
        int k = in.nextInt();
        System.out.println(describe(words, k));
        System.out.println(describe(ints, k));
```

```
    }
}
```

EXERCISE 2

max and reverse, for any type

Write two generic static methods: `<T extends Comparable<T>> T max(List<T> xs)` (the first maximum on ties) and `<T> void reverse(List<T> xs)` (in place, swapping from both ends). Read an integer `n` and `n` integers, then `m` and `m` words. Print `maxInt=<…>`, `maxWord=<…>` (natural String order), then the reversed lists as `ints=[…]` and `words=[…]`. One implementation each, used for both types.

Example: `3` then `4 9 2`, then `2` then `pear apple` →

```
maxInt=9
maxWord=pear
ints=[2, 9, 4]
words=[apple, pear]
```

INPUT 1

```
3
4 9 2
2
pear apple
```

OUTPUT 1

```
maxInt=9
maxWord=pear
ints=[2, 9, 4]
words=[apple, pear]
```

INPUT 2

```
1
7
1
solo
```

OUTPUT 2

```
maxInt=7
maxWord=solo
ints=[7]
words=[solo]
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static <T extends Comparable<T>> T max(List<T> xs) {
        T best = xs.get(0);
        for (T x : xs) if (x.compareTo(best) > 0) best = x;
        return best;
    }
    static <T> void reverse(List<T> xs) {
        for (int i = 0, j = xs.size() - 1; i < j; i++, j--) {
            T t = xs.get(i);
            xs.set(i, xs.get(j));
            xs.set(j, t);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> ints = new ArrayList<>();
        for (int i = 0; i < n; i++) ints.add(in.nextInt());
        int m = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < m; i++) words.add(in.next());
        System.out.println("maxInt=" + max(ints));
        System.out.println("maxWord=" + max(words));
        reverse(ints);
        reverse(words);
        System.out.println("ints=" + ints);
        System.out.println("words=" + words);
    }
}
```

QUIZ

1. Where is a generic method's type parameter declared?

A. After the method name

B. Before the return type: `static <T> T f(...)`

C. In the parameter list

D. In the class header

2. Inside `class Box<T>`, `static T shared;` is…

A. Fine

B. A compile error — the class's `T` cannot be used in a static context

C. Allowed if final

D. Allowed with a bound

3. `public <T> void confuse(T x)` inside `class Box<T>` declares…

A. The same `T`

B. A new, unrelated `T` that shadows the class parameter — a bug

C. A compile error

D. A bound on `T`

4. How does `ArrayList<E>` store its elements?

A. In an `E[]`

B. In an `Object[]`, casting to `E` on the way out

C. In a linked structure

D. In a `List<Object>`

5. `List<String> xs = Collections.emptyList();` infers `T` from…

A. The method body

B. The target type of the assignment

C. A default of `Object`

D. Nothing — it fails

ANSWER KEY · WRITING GENERIC CLASSES AND METHODS

1. B The `<T>` before the return type introduces the parameter for that method only.

2. B One erased class serves every `T`; a static member cannot depend on a per-instantiation type.

3. B Use a fresh letter (`U`) for method-level parameters in generic classes.

4. B `new E[n]` is impossible under erasure; the cast is contained and safe because only `add(E)` writes.

5. B Target typing lets generic methods with no arguments pick their type parameter from the assignment.

LESSON 3 · 12 MIN

Bounded type parameters

An unbounded `T` is any reference type, which means inside the method you can do nothing with it except store and return it — no `compareTo`, no `doubleValue`, no `close`. A **bound** restricts `T` to subtypes of a given type and, in exchange, lets you call that type's methods. Bounds are what make a generic `max`, a generic `sum` and a generic `sort` possible, and the recursive form `<T extends Comparable<T>>` is the one everyone stumbles over the first time.

Upper bounds with `extends`

```
public static <T extends Number> double sum(List<T> values) {
    double total = 0;
    for (T v : values) total += v.doubleValue();      // allowed: T is a Numbe
    return total;
}

sum(List.of(1, 2, 3));              // T = Integer
sum(List.of(1.5, 2.5));             // T = Double
sum(List.of("a"));                  // compile error: String is not within bou
```

`<T extends Number>` reads "T is `Number` or a subtype". `extends` is used whether the bound is a class or an interface — there is no `implements` in a bound. With the bound, every method of `Number` is available on a `T`; without it, only `Object`'s.

Under erasure, `T` compiles to its bound (`Number`) rather than `Object`, so the bytecode calls `Number.doubleValue` directly.

Multiple bounds

```
public static <T extends Number & Comparable<T>> T clampedMax(List<T> xs) { …
```

`&` joins bounds: `T` must satisfy all. At most one may be a class and it must come first; the rest are interfaces. Erasure uses the first bound. Multiple bounds are uncommon; when you need them, consider whether a small interface capturing the combination would read better.

`Comparable` and the recursive bound

The canonical generic `max`:

```
public static <T extends Comparable<T>> T max(List<T> xs) {
    T best = xs.get(0);
    for (T x : xs) if (x.compareTo(best) > 0) best = x;
    return best;
}
```

`<T extends Comparable<T>>` says: `T` can be compared **to itself**. `String implements Comparable<String>`, `Integer implements Comparable<Integer>`, your `Version implements Comparable<Version>` — all qualify. The bound *mentions the parameter it bounds*; that is a *recursive* (F-bounded) type parameter, and it looks alarming until you read it as "a T that knows how to compare with other Ts".

The JDK's real signature goes one step further:

```
public static <T extends Comparable<? super T>> T max(Collection<? extends T>
```

`? super T` allows a class that inherits its `Comparable` from a parent: if `Employee implements Comparable<Employee>` and `Manager extends Employee`, then `max(List<Manager>)` works because `Manager` is comparable *via* `Comparable<Employee>`, a supertype of `Manager`. Wildcards are the next lesson; note now that the flexible form is what library code uses.

Bounds on class type parameters

```
public class SortedBag<T extends Comparable<T>> {
    private final List<T> items = new ArrayList<>();
    public void add(T item) {
        int i = 0;
        while (i < items.size() && items.get(i).compareTo(item) < 0) i++;
        items.add(i, item);
    }
}
SortedBag<Integer> ints = new SortedBag<>();
SortedBag<Object> objs = new SortedBag<>();        // compile error: Object is
```

A class-level bound applies everywhere the class is instantiated — the compiler refuses `SortedBag<Object>`.

Bounds and inference

Bounds participate in inference: `max(List.of(3, 1, 2))` infers `T = Integer` and checks `Integer extends Comparable<Integer>`. A failed bound yields "inference variable T has incompatible bounds" — the message means the argument type does not satisfy the bound, not that inference is broken.

No lower bounds on type parameters

`<T super Integer>` does **not** exist. Lower bounds are only available on wildcards (`List<? super Integer>`) — next lesson — because a lower-bounded type *parameter* would rarely mean anything useful for the method body.

Bounds versus overloading versus `Object`

Three ways to write a method that works on many types:

- `Object` **parameter**: accepts anything, can do nothing type-specific, returns `Object` — pre-generics style.
- **Overloads** per type: `max(int[])`, `max(double[])`, `max(String[])` — repetitive; the JDK's `Math.max` is this for primitives, which cannot be generic.
- **Bounded generic**: one implementation, type-checked calls, typed return — the answer for reference types.

Reading a bounded signature

`static <K extends Comparable<K>, V> Map<K, V> sortedByKey(Map<K, V> m)` — two parameters; `K` must be self-comparable; `V` unconstrained; takes and returns a map with those parameters. Read bounds as constraints on *who may call*, and as *what the body may do* with the values.

Interview angle

"What does `<T extends Comparable<T>>` mean?" `T` must be a type that can be compared with other `T`s — so the body can call `compareTo`.

"Why `Comparable<? super T>` in the JDK's `max`?" To accept subclasses that inherit `Comparable` from a parent.

"Can you write `<T super Number>`?" No; lower bounds exist only on wildcards.

"Multiple bounds?" `<T extends A & B & C>` — one class (first), any interfaces.

"What does erasure turn a bounded `T` into?" Its first bound.

Key takeaways

`<T extends Bound>` restricts callers and unlocks the bound's methods in the body; `extends` for classes and interfaces alike.

`<T extends Comparable<T>>` is the self-comparable bound behind every generic `max` / `sort`; the JDK's `Comparable<? super T>` also admits inherited comparability.

`&` combines bounds (class first). No lower bounds on type parameters.

Bounds on class parameters restrict every instantiation.

EXERCISE 1

Bounded utilities

Write two bounded generic methods: `static <T extends Comparable<T>> T max(List<T> xs)` and `static <T extends Number> double average(List<T> xs)`. Read an integer `n`, `n` integers, then an integer `m` and `m` words. Print `max int=<max>`, `avg=<average with 2 decimals>`, `max word=<max word by compareTo>` — the same `max` serving `Integer` and `String`.

Example: `3` then `4 9 2`, `2` then `pear apple` →

```
max int=9
avg=5.00
max word=pear
```

INPUT 1

```
3
4 9 2
2
pear apple
```

OUTPUT 1

```
max int=9
avg=5.00
max word=pear
```

INPUT 2

```
1
-7
3
b a c
```

OUTPUT 2

```
max int=-7
avg=-7.00
max word=c
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static <T extends Comparable<T>> T max(List<T> xs) {
        T best = xs.get(0);
        for (T x : xs) if (x.compareTo(best) > 0) best = x;
        return best;
    }

    static <T extends Number> double average(List<T> xs) {
        double sum = 0;
        for (T x : xs) sum += x.doubleValue();
        return sum / xs.size();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> ints = new ArrayList<>();
        for (int i = 0; i < n; i++) ints.add(in.nextInt());
        int m = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < m; i++) words.add(in.next());
        System.out.println("max int=" + max(ints));
        System.out.printf("avg=%.2f%n", average(ints));
        System.out.println("max word=" + max(words));
    }
}
```

EXERCISE 2

Sum any numbers

Write `static <T extends Number> double sum(List<T> xs)` using `doubleValue()`, and `static <T extends Number & Comparable<T>> T clampMax(List<T> xs, T cap)` returning the largest element not exceeding `cap` (or `cap` itself if none). Read `n` integers and `m` decimals, then a cap integer. Print `sumInts=<…>`, `sumDoubles=<…>` (two decimals), `clamped=<clampMax(ints, cap)>`.

Example: `3` then `1 2 3`, `2` then `0.5 1.25`, then `2` →

```
sumInts=6.00
sumDoubles=1.75
clamped=2
```

INPUT 1

```
3
1 2 3
2
0.5 1.25
2
```

OUTPUT 1

```
sumInts=6.00
sumDoubles=1.75
clamped=2
```

INPUT 2

```
2
10 20
1
3.5
5
```

OUTPUT 2

```
sumInts=30.00
sumDoubles=3.50
clamped=5
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static <T extends Number> double sum(List<T> xs) {
        double s = 0;
        for (T x : xs) s += x.doubleValue();
        return s;
    }
    static <T extends Number & Comparable<T>> T clampMax(List<T> xs, T cap) {
        T best = null;
        for (T x : xs) {
            if (x.compareTo(cap) ⩽ 0 && (best == null || x.compareTo(best) > 0)) be
        }
        return best == null ? cap : best;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> ints = new ArrayList<>();
        for (int i = 0; i < n; i++) ints.add(in.nextInt());
        int m = in.nextInt();
        List<Double> doubles = new ArrayList<>();
        for (int i = 0; i < m; i++) doubles.add(in.nextDouble());
        int cap = in.nextInt();
        System.out.println(String.format(Locale.ROOT, "sumInts=%.2f", sum(ints)));
        System.out.println(String.format(Locale.ROOT, "sumDoubles=%.2f", sum(doubles
        System.out.println("clamped=" + clampMax(ints, cap));
    }
}
```

QUIZ

1. `<T extends Number>` allows the body to call…

A. Only `Object` methods

B. `Number`'s methods such as `doubleValue()` on any `T`

C. Any method

D. `compareTo`

2. `<T extends Comparable<T>>` means…

A. `T` extends a class named `Comparable`

B. `T` can be compared with other `T`s — the self-comparable bound

C. `T` must be a `String`

D. `T` is an interface

3. Why does the JDK use `Comparable<? super T>` rather than `Comparable<T>` in `sort`?

A. Style

B. So a subclass that inherits `Comparable<Parent>` from its parent still qualifies

C. It is faster

D. To allow nulls

4. `<T extends A & B>` requires…

A. `A` and `B` to be classes

B. At most one class (first), then interfaces

C. Exactly two interfaces

D. `A` to extend `B`

5. `<T super Number>` on a method is…

A. Valid

B. Invalid — lower bounds exist only on wildcards

C. The same as `extends`

D. Deprecated

ANSWER KEY · BOUNDED TYPE PARAMETERS

1. B The bound is what makes the type's methods available; without it `T` is effectively `Object`.

2. B The parameter appears in its own bound; read it as "a T that knows how to compare with Ts".

3. B `Manager extends Employee implements Comparable<Employee>` is comparable via a supertype's `Comparable`.

4. B Multiple bounds join with `&`; erasure uses the first bound.

5. B Only `? super X` is legal; a lower-bounded type parameter would rarely be useful for a method body.

LESSON 4 · 15 MIN

Wildcards and PECS

`List<Integer>` is not a `List<Number>`, even though `Integer` is a `Number`. That single fact — generics are **invariant** — is why wildcards exist: `List<? extends Number>` is a list of *some* subtype of `Number`, `List<? super Integer>` a list of *some* supertype. Which one to use is decided by whether you read from or write to the list, and the mnemonic **PECS** — Producer Extends, Consumer Super — makes it mechanical. This lesson covers the three wildcard forms, why invariance is necessary, and how to write flexible signatures like the JDK's.

Invariance, and why

```
List<Integer> ints = new ArrayList<>();
List<Number> nums = ints;              // compile error: incompatible types
```

Suppose it were allowed. Then `nums.add(3.14)` would put a `Double` into a list that `ints` still sees as `List<Integer>`, and `Integer i = ints.get(0)` would explode. Arrays permit exactly this (`Number[] n = new Integer[1]; n[0] = 3.14;` → `ArrayStoreException` at run time); generics were designed to catch it at compile time, and invariance is the price.

But some code genuinely does not care which subtype: a method that *sums* a list of numbers only reads them. Wildcards express that.

`? extends T` — a producer you can read from

```java
static double sum(List<? extends Number> xs) {
    double s = 0;
    for (Number n : xs) s += n.doubleValue();       // reading gives a Number
    return s;
}
sum(List.of(1, 2));          // List<Integer>  ✓
sum(List.of(1.5));           // List<Double>   ✓

List<? extends Number> ns = new ArrayList<Integer>();
ns.add(1);                   // compile error: what if ns is really a List<Dou
ns.add(null);                // the only thing you may add
Number n = ns.get(0);        // reading is fine
```

`List<? extends Number>` is a list whose element type is `Number` or some unknown subtype. You can **get** elements (as `Number`) but **cannot add** anything except `null`, because the compiler does not know which subtype the list really holds. It is a *producer* of `Number`s.

`? super T` — a consumer you can write to

```java
static void fillWithZeros(List<? super Integer> xs, int n) {
    for (int i = 0; i < n; i++) xs.add(0);          // adding an Integer is s
}
fillWithZeros(new ArrayList<Integer>(), 3);      // ✓
fillWithZeros(new ArrayList<Number>(), 3);       // ✓
fillWithZeros(new ArrayList<Object>(), 3);       // ✓

List<? super Integer> xs = new ArrayList<Number>();
Object o = xs.get(0);        // reading gives only Object — could be a Number,
```

`List<? super Integer>` holds `Integer` or some unknown supertype. You can **add** `Integer`s (they fit whatever the actual type is) but reads come back as `Object`. It is a *consumer* of `Integer`s.

PECS

Producer → `extends`; Consumer → `super`. Decide by what the *method* does with the parameter:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    for (T t : src) dest.add(t);                    // src produces Ts, dest c
}
public static <T extends Comparable<? super T>> void sort(List<T> list)     //
public static <T> T max(Collection<? extends T> coll, Comparator<? super T> co
Stream<R> map(Function<? super T, ? extends R> mapper)
```

If a parameter is *both* read and written (a list you sort in place), use a plain type parameter `List<T>` — no wildcard. Return types rarely use wildcards: `List<? extends Number>` as a return forces every caller to deal with the wildcard.

Unbounded `?`

```
static void printAll(List<?> xs) {
    for (Object o : xs) System.out.println(o);      // reads as Object; cannot
}
static int size(Collection<?> c) { return c.size(); }
if (obj instanceof List<?> list) …                  // the only generic form i
```

`List<?>` is "a list of something": use it when the element type is irrelevant to the method (size, printing, clearing) or when you must name a generic type without committing (`Class<?>` , `List<?>` in `instanceof`). `List<?>` is *not* `List<Object>` — the latter accepts only actual `List<Object>` arguments; the former accepts any list.

Wildcard capture

```
static void swapFirstTwo(List<?> list) {
    swapHelper(list);                                // the compiler CAPTURES
}
private static <T> void swapHelper(List<T> list) {   // now we can read and wr
    T tmp = list.get(0);
    list.set(0, list.get(1));
    list.set(1, tmp);
}
```

A method taking `List<?>` cannot write to it — but it can delegate to a generic helper, and the compiler *captures* the unknown type as a named parameter for that call. "Capture of ?" in

error messages refers to this; the helper pattern is the standard resolution.

Reading JDK signatures now

`Collections.max`: `<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)` — a collection producing `T`s (so `extends`), where `T` is comparable to itself or an ancestor (so `super`). Every wildcard in the JDK follows PECS; once you see the pattern, the signatures read themselves.

When to skip wildcards

Local variables: `List<Integer> xs`, never `List<? extends Integer> xs`.

Return types: concrete.

Parameters you both read and write: `List<T>`.

Simple private helpers where flexibility buys nothing.

Wildcards belong on **public API parameters** where callers might reasonably pass a `List<Integer>` to a method conceptually about `Number`s.

Interview angle

"Why isn't `List<Integer>` a `List<Number>`?" Invariance: otherwise you could add a `Double` through the `Number` view.

"Difference between `List<?>` and `List<Object>`?" `List<?>` accepts any list and allows only reads as `Object`; `List<Object>` accepts only lists declared as `List<Object>`.

"What is PECS?" Producer `extends`, consumer `super` — choose the wildcard by whether you read from or write to the parameter.

"Can you add to a `List<? extends Number>`?" Only `null`.

"What is wildcard capture?" The compiler naming an unknown `?` as a fresh type parameter for a generic helper call.

Key takeaways

Generics are invariant; wildcards add controlled variance.

`? extends T`: read as `T`, cannot add — producers. `? super T`: add `T`, read as `Object` — consumers. PECS.

`?` alone when the element type does not matter; it is not `Object`.

Wildcards on API parameters; plain parameters for read-and-write and for locals/returns.

Capture unknowns with a generic helper when you need to write.

EXERCISE 1

PECS in practice

Write `static double sum(Collection<? extends Number> xs)` (a producer), `static void addRange(Collection<? super Integer> dest, int from, int to)` (a consumer that adds `from..to` inclusive), and `static <T> void copy(List<? super T> dest, List<? extends T> src)`.

Read integers `from` and `to`. Add the range to a `List<Number>`, then also add `0.5` to that list; print `sum=<sum with 1 decimal>`. Copy a `List<Integer>` of `from..to` into a `List<Object>` with `copy` and print `copied=<dest size>`.

Example: `1 4` →

```
sum=10.5
copied=4
```

INPUT 1

```
1 4
```

OUTPUT 1

```
sum=10.5
copied=4
```

INPUT 2

```
5 5
```

OUTPUT 2

```
sum=5.5
copied=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static double sum(Collection<? extends Number> xs) {
        double s = 0;
        for (Number n : xs) s += n.doubleValue();
        return s;
    }

    static void addRange(Collection<? super Integer> dest, int from, int to) {
        for (int i = from; i <= to; i++) dest.add(i);
    }

    static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (T t : src) dest.add(t);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int from = in.nextInt(), to = in.nextInt();
        List<Number> numbers = new ArrayList<>();
        addRange(numbers, from, to);
        numbers.add(0.5);
        System.out.printf("sum=%.1f%n", sum(numbers));

        List<Integer> ints = new ArrayList<>();
        addRange(ints, from, to);
        List<Object> objects = new ArrayList<>();
        copy(objects, ints);
        System.out.println("copied=" + objects.size());
    }
}
```

EXERCISE 2

Copy with PECS

Write `static <T> int copyInto(List<? super T> dst, List<? extends T> src)` that appends every element of `src` to `dst` and returns how many were copied, and `static double total(Collection<? extends Number> xs)`. Read `n` integers into a `List<Integer>` and `m` doubles into a `List<Double>`. Copy both into one `List<Number>` with `copyInto`, then print `copied=<count>`, `numbers=<the list>` and `total=<total(numbers) to two decimals>`. The wildcards are what let a `List<Integer>` flow into a `List<Number>`.

Example: `2` then `1 2`, `1` then `0.5` →

```
copied=3
numbers=[1, 2, 0.5]
total=3.50
```

INPUT 1

```
2
1 2
1
0.5
```

OUTPUT 1

```
copied=3
numbers=[1, 2, 0.5]
total=3.50
```

INPUT 2

```
0
2
1.5 2.5
```

OUTPUT 2

```
copied=2
numbers=[1.5, 2.5]
total=4.00
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static <T> int copyInto(List<? super T> dst, List<? extends T> src) {
        for (T x : src) dst.add(x);
        return src.size();
    }
    static double total(Collection<? extends Number> xs) {
        double s = 0;
        for (Number x : xs) s += x.doubleValue();
        return s;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> ints = new ArrayList<>();
        for (int i = 0; i < n; i++) ints.add(in.nextInt());
        int m = in.nextInt();
        List<Double> doubles = new ArrayList<>();
        for (int i = 0; i < m; i++) doubles.add(in.nextDouble());
        List<Number> numbers = new ArrayList<>();
        int copied = copyInto(numbers, ints) + copyInto(numbers, doubles);
        System.out.println("copied=" + copied);
        System.out.println("numbers=" + numbers);
        System.out.println(String.format(Locale.ROOT, "total=%.2f", total(numbers)))
    }
}
```

QUIZ

1. `List<Number> nums = new ArrayList<Integer>();` fails because…

A. `Integer` is not a `Number`

B. Generics are invariant: allowing it would let a `Double` be added through `nums`

C. `ArrayList` is final

D. The diamond is missing

2. Into a `List<? extends Number>` you may add…

A. Any `Number`

B. Only `null`

C. Only `Integer`

D. Anything

3. Reading from a `List<? super Integer>` yields…

A. `Integer`

B. `Object`

C. `Number`

D. A compile error

4. PECS stands for…

A. Parameter Extends, Class Super

B. Producer Extends, Consumer Super

C. Public Extends, Concrete Super

D. Primitive Extends, Complex Super

5. `List<?>` differs from `List<Object>` in that…

A. Nothing

B. `List<?>` accepts any list (read-only as `Object`); `List<Object>` accepts only lists declared as `List<Object>`

C. `List<?>` allows writes

D. `List<Object>` is raw

ANSWER KEY · WILDCARDS AND PECS

1. B Invariance is what keeps the `Integer` list honest. Use `List<? extends Number>` for a read-only view.

2. B The actual element type is unknown (could be `Double`), so nothing but `null` is provably safe.

3. B The list could hold any supertype of `Integer`; only `Object` is guaranteed.

4. B Read from it → `? extends T`; write to it → `? super T`; both → plain `T`.

5. B The unbounded wildcard is the way to say "a list of something".

LESSON 5 · 14 MIN

Type erasure and its consequences

Everything the compiler knows about `List<String>` it forgets before the class file is written. That decision — **erasure** — is why Java generics are backward compatible, and why a whole list of natural-looking things are impossible: `new T()`, `T[]`, `instanceof List<String>`, overloading on type arguments, a static field of type `T`. This lesson explains what erasure does, walks through each limitation with the reason, and covers the two artefacts it leaves behind — bridge methods and heap pollution.

What erasure does

The compiler:

- **Checks** every generic use for type correctness.
- **Replaces** each type parameter by its erasure: `Object` for an unbounded `T`, the first bound for a bounded one (`<T extends Number>` → `Number`).
- **Inserts casts** where generic code returns a `T` to typed code: `String s = list.get(0)` becomes `String s = (String) list.get(0)`.
- **Generates bridge methods** where erasure would break overriding (below).

After that, `List<String>`, `List<Integer>` and raw `List` are the same class at run time. A JVM from 2004 can run today's generic code.

```
new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass()
List<String> a = …; a.getClass().getName()
```

Contrast C#, whose generics are *reified*: `List<string>` and `List<int>` are distinct run-time types. Java chose compatibility; the limitations below are the cost.

What you cannot do, and why

1. Instantiate a type parameter

```
T t = new T();              // error: unexpected type
```

`T` is `Object` at run time; there is no class to instantiate. Fix: pass a `Supplier<T>` (`factory.get()`) or a `Class<T>` token (`clazz.getDeclaredConstructor().newInstance()`).

2. Create a generic array

```
T[] arr = new T[10];                    // error: generic array creation
List<String>[] lists = new List<String>[3];   // error too
```

Arrays check their element type at run time (`ArrayStoreException`); a `T[]` would have no type to check against. Fix: `(T[]) new Object[10]` with a contained `@SuppressWarnings("unchecked")` (what `ArrayList` does), or `Array.newInstance(clazz, 10)` with a class token, or use a `List<T>`.

3. Test a type argument at run time

```
if (obj instanceof List<String>) …      // error: illegal generic type for ins
if (obj instanceof List<?>) …           // fine — the raw class is all that ex
List<String> l = (List<String>) obj;    // compiles with an UNCHECKED warning.
```

The JVM cannot see the `String`; the cast is a promise, not a check. If the promise is wrong, the `ClassCastException` appears later, at the first `String s = l.get(0)`.

4. Use a type parameter in a static context

```
class Box<T> { static T shared; }       // error
```

One class serves every `T`; a static field would have to be all of them at once.

5. Overload on type arguments

```
void process(List<String> xs) { }
void process(List<Integer> xs) { }      // error: name clash — both erase to p
```

Same erasure, same signature. Different method names, or one generic method.

6. Catch or throw a generic exception

```
class MyException<T> extends Exception { }     // error: a generic class may n
```

`catch` clauses are resolved by run-time class; a parameterised exception could not be distinguished.

7. Primitives as type arguments — `List<int>`: nothing to erase to. Wrappers.

Bridge methods

```
class Node<T> { T value; void set(T v) { value = v; } }
class IntNode extends Node<Integer> {
    @Override void set(Integer v) { … }         // overrides set(T)? After era
}
```

`set(Integer)` does not have the same erased signature as `set(Object)`, so the compiler generates a hidden **bridge** `set(Object v) { set((Integer) v); }` in `IntNode` so overriding works through the erased parent. You see bridges in stack traces and reflection (`Method.isBridge()`); you never write them. They are also why a `ClassCastException` can appear *inside* a bridge when heap pollution (next) delivers the wrong type.

Heap pollution

A variable of a parameterised type refers to an object that is not of that type:

```
List<String> strings = new ArrayList<>();
List raw = strings;
raw.add(42);                                // warning: unchecked call — the St
String s = strings.get(0);                  // ClassCastException here, with no
```

Raw types, unchecked casts and generic varargs are the three ways to cause it. Every "unchecked" warning marks a place where erasure lets a lie through; treat them as errors, and confine the unavoidable ones (`(T[]) new Object[n]`) to the smallest scope with `@SuppressWarnings("unchecked")` and a comment.

Reifiable types

A type whose full information survives erasure is **reifiable**: primitives, non-generic classes, raw types, `List<?>` (unbounded wildcard), arrays of reifiable types. Only reifiable types may appear in `instanceof`, array creation and `catch`. Everything with a concrete type argument is non-reifiable.

Type tokens: carrying the type at run time

```
static <T> T fromJson(String json, Class<T> type) { … type.cast(parsed) … }
User u = fromJson(text, User.class);

static <T> T[] newArray(Class<T> type, int n) { return (T[]) Array.newInstance
```

A `Class<T>` parameter reintroduces the type at run time: it can create instances, arrays, and checked casts (`type.cast(obj)` throws a *real* `ClassCastException` immediately). Libraries that need `List<User>` (not just `User`) use a *super type token* — an anonymous subclass of a generic class whose superclass type argument is retained by reflection (`new TypeReference<List<User>>() {}` in Jackson) — the one place generic information does survive.

Reflection and generic signatures

Erasure removes type arguments from *bytecode instructions*, but declarations keep a `Signature` attribute: `Field.getGenericType()` returns `List<String>` for a field declared that way, and `Class.getGenericSuperclass()` is what super type tokens read. Declarations remember; values do not.

Interview angle

"What is type erasure?" The compiler replaces type parameters with their bounds and inserts casts; the JVM sees raw types.

"Why can't you write `new T()`?" At run time `T` is `Object`; there is no class to construct. Use a `Supplier<T>` or `Class<T>`.

"Why can't you overload `f(List<String>)` and `f(List<Integer>)`?" Both erase to `f(List)`.

"What is a bridge method?" A compiler-generated method that lets an override with a specialised signature satisfy the erased parent signature.

"What is heap pollution?" A parameterised variable holding an object of the wrong type argument, via raw types or unchecked casts.

Key takeaways

Erasure: check at compile time, erase to `Object`/bound, insert casts. `List<String>` is `List` at run time.

Consequences: no `new T()`, no `T[]`, no `instanceof List<String>`, no static `T`, no overloading on type arguments, no generic exceptions, no primitives.

Bridge methods keep overriding working; heap pollution is the lie unchecked warnings point at.

Class tokens (`Class<T>`) and super type tokens carry type information into run time when you need it.

EXERCISE 1

Class tokens beat erasure

Write `static <T> T[] filled(Class<T> type, int n, Supplier<T> factory)` that creates a real `T[]` with `java.lang.reflect.Array.newInstance` and fills it from the supplier — the two things erasure forbids directly (`new T[n]`, `new T()`). Then write `static <T> int countOfType(Object[] items, Class<T> type)` using `type.isInstance`.

Read an integer `n`. Build a `String[]` of `n` elements `"s<index>"` and an `Integer[]` of `n` squares with `filled`; print both with `Arrays.toString`, then `strings=<count of String in a mixed Object[] made of both arrays>` and `array type=<the String[]'s getClass().getSimpleName()>`.

Example: `3` →

```
[s0, s1, s2]
[0, 1, 4]
strings=3
array type=String[]
```

INPUT 1

```
3
```

OUTPUT 1

```
[s0, s1, s2]
[0, 1, 4]
strings=3
array type=String[]
```

INPUT 2

```
0
```

OUTPUT 2

```
[]
[]
strings=0
array type=String[]
```

REFERENCE SOLUTION

```
import java.lang.reflect.Array;
import java.util.*;
import java.util.function.*;

public class Main {
    @SuppressWarnings("unchecked")
    static <T> T[] filled(Class<T> type, int n, Supplier<T> factory) {
        T[] arr = (T[]) Array.newInstance(type, n);
        for (int i = 0; i < n; i++) arr[i] = factory.get();
        return arr;
    }

    static <T> int countOfType(Object[] items, Class<T> type) {
        int c = 0;
        for (Object o : items) if (type.isInstance(o)) c++;
        return c;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] counter = {0};
        String[] strings = filled(String.class, n, () -> "s" + counter[0]++);
        int[] sq = {0};
        Integer[] squares = filled(Integer.class, n, () -> { int i = sq[0]++; returr
        System.out.println(Arrays.toString(strings));
        System.out.println(Arrays.toString(squares));
        Object[] mixed = new Object[2 * n];
        System.arraycopy(strings, 0, mixed, 0, n);
        System.arraycopy(squares, 0, mixed, n, n);
        System.out.println("strings=" + countOfType(mixed, String.class));
        System.out.println("array type=" + strings.getClass().getSimpleName());
    }
}
```

EXERCISE 2

Erasure, demonstrated

Read `n` words. Show what erasure means at run time: print `sameClass=<new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass()>`, `className=<that class's simple name>`. Then pollute the heap: put the words in a `List<String>`, assign it to a raw `List`, `add(42)` through the raw reference, and read the elements back through the typed reference one by one — print each string, and when the `ClassCastException` fires print `heap pollution at index <i>`. Finally print `size=<list size>`: the integer is really in there.

Example: `2` then `a b` →

```
sameClass=true
className=ArrayList
a
b
heap pollution at index 2
size=3
```

INPUT 1

```
2
a b
```

OUTPUT 1

```
sameClass=true
className=ArrayList
a
b
heap pollution at index 2
size=3
```

INPUT 2

```
0
```

OUTPUT 2

```
sameClass=true
className=ArrayList
heap pollution at index 0
size=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    @SuppressWarnings({"rawtypes", "unchecked"})
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        System.out.println("sameClass=" + (new ArrayList<String>().getClass() == new
        System.out.println("className=" + new ArrayList<String>().getClass().getSimp

        List raw = words;
        raw.add(42);
        for (int i = 0; i < words.size(); i++) {
            try {
                String s = words.get(i);
                System.out.println(s);
            } catch (ClassCastException e) {
                System.out.println("heap pollution at index " + i);
            }
        }
        System.out.println("size=" + words.size());
    }
}
```

QUIZ

1. After compilation, `List<String>` in bytecode is...

A. `List<String>`

B. Raw `List` — the argument is erased

C. `List<Object>`

D. `ArrayList`

2. `new T()` fails because…

A. Constructors cannot be generic

B. At run time `T` is `Object` — there is no class to construct; pass a `Supplier<T>` or `Class<T>`

C. `T` is abstract

D. It needs `new T<>()`

3. `obj instanceof List<String>` is…

A. Valid

B. A compile error — only reifiable types such as `List<?>` may be tested

C. Valid but always false

D. Valid at run time only

4. A bridge method is…

A. A default method

B. A compiler-generated method that makes an override with a specialised signature satisfy the erased parent signature

C. A static factory

D. A lambda

5. Heap pollution is…

A. A memory leak

B. A parameterised variable referring to an object of the wrong type argument, via raw types or unchecked casts

C. Too many objects on the heap

D. A garbage-collection failure

ANSWER KEY · TYPE ERASURE AND ITS CONSEQUENCES

1. B Erasure removes type arguments; casts are inserted where typed code reads elements.

2. B The type is gone by the time the code runs.

3. B The JVM cannot see the `String`; `instanceof List<?>` is the allowed form.

4. B `set(Integer)` overriding `set(T)` needs a synthetic `set(Object)` that delegates to it.

5. B The `ClassCastException` then appears far from the unchecked operation that caused it.

LESSON 6 · 12 MIN

Generics in practice — idioms and error messages

The previous lessons gave the rules; this one is the everyday texture: the idioms experienced Java code uses with generics, the compiler messages you will actually see and what each means, and the judgement calls — when to make something generic, when not to, how much wildcard is too much.

Idioms

Generic value containers. `Optional<T>`, `Pair<A, B>` (a record), `Result<T>` (a sealed interface with `Ok<T>` / `Err<T>` records) — small generic types that carry a value with a meaning. Records make these one-liners: `record Pair<A, B>(A first, B second) {}`.

Generic utility methods on a `final` class with a private constructor:

```
public final class Lists {
    private Lists() {}
    public static <T> List<T> reversed(List<T> xs) { … }
    public static <T, R> List<R> map(List<T> xs, Function<? super T, ? extends
    public static <T extends Comparable<? super T>> T max(Collection<? extends
}
```

Typed empty results: `List.of()`, `Collections.emptyList()`, `Optional.empty()` infer their type from the target; return them instead of `null`.

Comparator building relies entirely on generic inference:

```
Comparator<Person> byAge = Comparator.comparingInt(Person::age);
Comparator<Person> byName = Comparator.comparing(Person::lastName, String.CASE
people.sort(byAge.thenComparing(byName).reversed());
```

When inference fails in a chain (`Comparator.comparing(Person::lastName).thenComparing(…)` complaining that `Object` has no `lastName`), give the first lambda an explicit parameter type: `Comparator.comparing((Person p) → p.lastName())`. The chain's first call has no target type to infer from.

Class tokens for run-time types: `Class<T>` parameters (`Enum.valueOf(Day.class, s)`, `EnumMap`, `fromJson(text, User.class)`).

Bounded generic algorithms: `<T extends Comparable<? super T>>` on anything that orders; `<T extends Number>` on anything that computes.

The compiler messages, decoded

Message	Meaning	Fix
`incompatible types: List<Integer> cannot be converted to List<Number>`	Invariance	Use `List<? extends Number>` in the parameter
`unchecked call to add(E) as a member of the raw type List`	A raw type is in use	Add the type arguments
`unchecked cast`	`(List<String>) obj` — cannot be verified	Restructure; or contain with `@SuppressWarnings("unchecked")` and a justification
`generic array creation`	`new T[n]` / `new List<String>[n]`	`Object[]` with cast, `Array.newInstance`, or a `List`
`non-static type variable T cannot be referenced from a static context`	Class parameter used in a static member	Declare `<T>` on the static method
`inference variable T has incompatible bounds`	Arguments imply conflicting `T`s	Check the argument types; add a witness `Util.<String>f(…)`
`name clash: f(List<String>) and f(List<Integer>) have the same erasure`	Overloading on type arguments	Rename one
`capture#1 of ? extends Number` in a message	You tried to write into a `? extends` list	Use `? super`, a plain `T`, or a capture helper
`possible heap pollution from parameterized vararg type`	Generic varargs	`@SafeVarargs` if the array is only read
`cannot select from a type variable` (`T.class`, `T.CONSTANT`)	No static access through `T`	Pass a `Class<T>`

Each message points at one rule from earlier lessons; recognising which one is most of the debugging.

`@SuppressWarnings("unchecked")` — the discipline

Unchecked warnings are the compiler admitting it cannot verify safety. Rules for suppressing:

Prove to yourself the code is safe (usually: the only writes are typed).

Suppress on the **smallest** element — a local variable declaration or a single method, never a class.

Comment why it is safe.

```
@SuppressWarnings("unchecked")
T item = (T) items[--size];     // safe: only push(T) stores into items
```

Unexplained suppressions are where heap pollution hides.

How generic to be

Make it generic when the same code would otherwise be duplicated per type, or when the element type is part of the contract (`Repository<T>` , `Cache<K, V>` , `Result<T>`).

Do not make a class generic to look sophisticated: a `Config<T>` where `T` is always `String` is noise.

Wildcards on public parameters where callers benefit (`Collection<? extends T>`); not on locals, fields or returns.

Bounds only when the body needs the bound's methods.

A single type parameter covers most cases; three or more suggests a missing abstraction.

Generics and collections, together

The collections module (next) is generics in daily use: `Map<String, List<Order>>` , `Set<? extends Shape>` , `Deque<Character>` , `Comparator<? super T>` . The idioms there — `computeIfAbsent(k, key → new ArrayList<>())` , `List.copyOf` , `Collections.unmodifiableList` — all depend on inference working as this module described.

Generics and lambdas, together

Functional interfaces are generic (`Function<T, R>`), so every lambda's type is a parameterised type inferred from context. When a stream pipeline's types go wrong, the first `map` is usually where inference lost the thread; typing the lambda parameter (`(String s) → …`) restores it.

A worked example: a generic bounded cache

```
public final class LruCache<K, V> {
    private final LinkedHashMap<K, V> map;
    public LruCache(int capacity) {
        map = new LinkedHashMap<>(16, 0.75f, true) {          // access-order
            @Override protected boolean removeEldestEntry(Map.Entry<K, V> elde
        };
    }
    public V get(K key) { return map.get(key); }
    public void put(K key, V value) { map.put(key, value); }
    public V computeIfAbsent(K key, Function<? super K, ? extends V> loader) {
}
```

Two class parameters, an anonymous subclass that sees them (it is an inner class), and a PECS-correct `Function<? super K, ? extends V>` on the one public method that takes a function. This is what "generics in practice" looks like: a few lines, all of them mechanical once the rules are known.

Interview angle

"How do you handle an unchecked cast you cannot avoid?" Prove it safe, suppress on the smallest scope, comment the reason.

"Why does `Comparator.comparing(Person::name).thenComparing(…)` sometimes fail to compile?" The first call has no target type; give the lambda an explicit parameter type.

"When would you not use generics?" When the type is always the same, or when the parameter would not be used in the body.

"What does `capture of ?` mean in an error?" You tried to write to a wildcard-typed collection; use `super` or a helper.

Key takeaways

Idioms: generic records for values, static utility methods with bounds and PECS, typed empties, class tokens, comparator chains.

Learn the error messages — each maps to one rule (invariance, erasure, statics, inference).

Suppress unchecked warnings narrowly, with proof and a comment.

Be as generic as the contract requires and no more; wildcards on API parameters only.

EXERCISE 1

A generic LRU cache

Write `class LruCache<K, V>` backed by an access-ordered `LinkedHashMap` (constructor `new LinkedHashMap<>(16, 0.75f, true)`) whose anonymous subclass overrides `removeEldestEntry` to evict beyond a capacity. Provide `V get(K key)` (returns null if absent), `void put(K key, V value)`, `V computeIfAbsent(K key, Function<? super K, ? extends V> loader)` and `Set<K> keys()` (the map's key set, in access order).

Read a capacity `c`, an integer `n` and `n` commands: `put <k> <v>`, `get <k>` (print the value or `miss`), `load <k>` (`computeIfAbsent` with the loader `k → k.toUpperCase()`; print the value). After all commands print `keys=<keys joined by ,>`.

Example: capacity `2`, then `put a 1`, `put b 2`, `get a`, `put c 3`, `get b` →

```
1
miss
keys=a,c
```

INPUT 1

```
2
5
put a 1
put b 2
get a
put c 3
get b
```

OUTPUT 1

```
1
miss
keys=a,c
```

INPUT 2

```
1
3
load x
load y
get x
```

OUTPUT 2

```
X
Y
miss
keys=y
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.function.*;

class LruCache<K, V> {
    private final LinkedHashMap<K, V> map;

    LruCache(int capacity) {
        map = new LinkedHashMap<>(16, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry<K, V> eldest) { return siz
        };
    }

    V get(K key) { return map.get(key); }
    void put(K key, V value) { map.put(key, value); }
    V computeIfAbsent(K key, Function<? super K, ? extends V> loader) { return map.c
    Set<K> keys() { return map.keySet(); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int c = in.nextInt();
        int n = in.nextInt();
        LruCache<String, String> cache = new LruCache<>(c);
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "put" -> cache.put(in.next(), in.next());
                case "get" -> {
                    String v = cache.get(in.next());
                    System.out.println(v == null ? "miss" : v);
                }
                default -> System.out.println(cache.computeIfAbsent(in.next(), k ->
            }
        }
        System.out.println("keys=" + String.join(",", cache.keys()));
    }
}
```

EXERCISE 2

argmax over any map

Write `static <K, V extends Comparable<? super V>> K argmax(Map<K, V> m)` returning the key with the largest value (the first met in iteration order on ties) and `static <K, V extends`

`Comparable<? super V>> List<K> keysAbove(Map<K, V> m, V threshold)` returning keys whose value exceeds the threshold, in iteration order. Read `n` lines `<name> <score>` into a `LinkedHashMap<String, Integer>` and a threshold; print `best=<argmax>` and `above=<keysAbove>`.

Example: `3` then `ann 70`, `bob 90`, `cy 90`, then `75` →

```
best=bob
above=[bob, cy]
```

INPUT 1

```
3
ann 70
bob 90
cy 90
75
```

OUTPUT 1

```
best=bob
above=[bob, cy]
```

INPUT 2

```
1
solo 1
1
```

OUTPUT 2

```
best=solo
above=[]
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static <K, V extends Comparable<? super V>> K argmax(Map<K, V> m) {
        K best = null;
        V bestValue = null;
        for (Map.Entry<K, V> e : m.entrySet()) {
            if (bestValue == null || e.getValue().compareTo(bestValue) > 0) {
                best = e.getKey();
                bestValue = e.getValue();
            }
        }
        return best;
    }
    static <K, V extends Comparable<? super V>> List<K> keysAbove(Map<K, V> m, V thr
        List<K> out = new ArrayList<>();
        for (Map.Entry<K, V> e : m.entrySet()) if (e.getValue().compareTo(threshold)
        return out;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, Integer> scores = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) scores.put(in.next(), in.nextInt());
        int threshold = in.nextInt();
        System.out.println("best=" + argmax(scores));
        System.out.println("above=" + keysAbove(scores, threshold));
    }
}
```

QUIZ

1. `@SuppressWarnings("unchecked")` should be placed…

A. On the class

B. On the smallest scope possible, with a comment explaining why the cast is safe

C. On `main`

D. Nowhere — always fix the warning

2. `Comparator.comparing(Person::name).thenComparing(Person::age)` fails to compile with "cannot find symbol". The fix is…

A. Use a loop

B. Give the first lambda an explicit parameter type: `comparing((Person p) → p.name())`

C. Remove `thenComparing`

D. Add `@SuppressWarnings`

3. Wildcards belong mainly on…

A. Local variables

B. Return types

C. Public API parameters where callers benefit from flexibility

D. Fields

4. A `Class<T>` parameter lets a generic method…

A. Skip erasure

B. Create instances and arrays of `T` and perform checked casts at run time

C. Use primitives

D. Overload on `T`

5. A class `Config<T>` where `T` is always `String` in practice is…

A. Good design

B. Needless generality — make it concrete

C. Required by frameworks

D. Faster

ANSWER KEY · GENERICS IN PRACTICE — IDIOMS AND ERROR MESSAGES

1. B Some casts are unavoidable under erasure; confine and justify them so heap pollution has nowhere to hide.

2. B The first call in a chain has no target type to infer `T` from; an explicit parameter type restores inference.

3. C Locals, fields and returns should be concrete; wildcards in a return type burden every caller.

4. B The class token carries the type information erasure removed.

5. B Be as generic as the contract requires; unused flexibility is noise.

CHECKPOINT · 24 MIN

Checkpoint — Generics

This checkpoint covers the purpose of generics, generic classes and methods, bounded type parameters, wildcards and PECS, type erasure and its consequences, and the practical idioms.

How it works. Fourteen questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- Why `List<Integer>` is not a `List<Number>`, and which wildcard fixes a read-only parameter.
- What `<T extends Comparable<? super T>>` means, word by word.
- Which of `new T()`, `T[]`, `instanceof List<String>`, `static T` are legal, and why not.
- What PECS stands for and how to apply it to `copy(dest, src)`.
- What a raw type does to type safety.

The programs are a generic bounded container with a PECS-correct method, and a generic utility set whose signatures the compiler must accept for several element types.

EXERCISE 1

A bounded box with PECS methods

Write `class MinMax<T extends Comparable<? super T>>` that keeps the smallest and largest of the values offered to it: `void offer(T value)`, `void offerAll(Collection<? extends T> values)`, `void drainTo(Collection<? super T> dest)` (adds min then max, if any, and resets), and `Optional<T> min()` / `max()`.

Read an integer `n` and `n` integers; offer them all through `offerAll` to a `MinMax<Integer>`; print `min=<v> max=<v>` (or `empty` if n is 0); drain into a `List<Number>` and print its contents with `toString`, then `after drain=<min().isPresent()>`.

Example: `3` then `5 -2 9` →

```
min=-2 max=9
[-2, 9]
after drain=false
```

INPUT 1

```
3
5 -2 9
```

OUTPUT 1

```
min=-2 max=9
[-2, 9]
after drain=false
```

INPUT 2

```
0
```

OUTPUT 2

```
empty
[]
after drain=false
```

REFERENCE SOLUTION

```java
import java.util.*;

class MinMax<T extends Comparable<? super T>> {
    private T min, max;

    void offer(T value) {
        if (min == null || value.compareTo(min) < 0) min = value;
        if (max == null || value.compareTo(max) > 0) max = value;
    }

    void offerAll(Collection<? extends T> values) {
        for (T v : values) offer(v);
    }

    void drainTo(Collection<? super T> dest) {
        if (min != null) {
            dest.add(min);
            dest.add(max);
        }
        min = null;
        max = null;
    }

    Optional<T> min() { return Optional.ofNullable(min); }
    Optional<T> max() { return Optional.ofNullable(max); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> values = new ArrayList<>();
        for (int i = 0; i < n; i++) values.add(in.nextInt());
        MinMax<Integer> mm = new MinMax<>();
        mm.offerAll(values);
        if (mm.min().isPresent()) System.out.println("min=" + mm.min().get() + " max
        else System.out.println("empty");
        List<Number> out = new ArrayList<>();
        mm.drainTo(out);
        System.out.println(out);
        System.out.println("after drain=" + mm.min().isPresent());
    }
}
```

EXERCISE 2

Generic utilities that compile for everything

Write three generic utilities: `static <T> List<T> reversed(List<? extends T> xs)` returning a new list; `static <T, R> List<R> mapAll(List<? extends T> xs, Function<? super T, ? extends R> f)`; and `static <K, V extends Comparable<? super V>> K argMax(Map<K, V> m)` returning the key with the largest value (any one on ties; `null` for an empty map).

Read an integer `n` and `n` words. Print `reversed(words)`, then `mapAll(words, String::length)`, then `argMax` of a map from each word to its length (build it with a `LinkedHashMap` and print `longest=<key>`).

Example: `3` then `pear fig banana` →

```
[banana, fig, pear]
[4, 3, 6]
longest=banana
```

INPUT 1

```
3
pear fig banana
```

OUTPUT 1

```
[banana, fig, pear]
[4, 3, 6]
longest=banana
```

INPUT 2

```
1
x
```

OUTPUT 2

```
[x]
[1]
longest=x
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;

public class Main {
    static <T> List<T> reversed(List<? extends T> xs) {
        List<T> out = new ArrayList<>(xs);
        Collections.reverse(out);
        return out;
    }

    static <T, R> List<R> mapAll(List<? extends T> xs, Function<? super T, ? extends
        List<R> out = new ArrayList<>(xs.size());
        for (T x : xs) out.add(f.apply(x));
        return out;
    }

    static <K, V extends Comparable<? super V>> K argMax(Map<K, V> m) {
        K bestKey = null;
        V bestValue = null;
        for (Map.Entry<K, V> e : m.entrySet()) {
            if (bestValue == null || e.getValue().compareTo(bestValue) > 0) {
                bestKey = e.getKey();
                bestValue = e.getValue();
            }
        }
        return bestKey;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());
        System.out.println(reversed(words));
        System.out.println(mapAll(words, String::length));
        Map<String, Integer> lengths = new LinkedHashMap<>();
        for (String w : words) lengths.put(w, w.length());
        System.out.println("longest=" + argMax(lengths));
    }
}
```

EXERCISE 3

Merge intervals of any comparable type

Write `record Interval<T extends Comparable<? super T>>(T lo, T hi)` and `static <T extends Comparable<? super T>> List<Interval<T>> merge(List<Interval<T>> xs)` that sorts by `lo` and

merges overlapping or touching intervals (`next.lo ⩽ current.hi`). Read `n` integer intervals and `m` word intervals (inclusive ranges of words in natural order), merge each list with the same method, and print them as `[lo, hi]` space-separated on one line each, then `merged=<count of integer intervals>+<count of word intervals>` .

Example: `3` then `1 3` , `2 6` , `8 10` , then `2` then `a c` , `b d` →

```
[1, 6] [8, 10]
[a, d]
merged=2+1
```

INPUT 1

```
3
1 3
2 6
8 10
2
a c
b d
```

OUTPUT 1

```
[1, 6] [8, 10]
[a, d]
merged=2+1
```

INPUT 2

```
2
5 5
1 2
1
m m
```

OUTPUT 2

```
[1, 2] [5, 5]
[m, m]
merged=2+1
```

REFERENCE SOLUTION

```java
import java.util.*;

record Interval<T extends Comparable<? super T>>(T lo, T hi) {
    @Override public String toString() { return "[" + lo + ", " + hi + "]"; }
}

public class Main {
    static <T extends Comparable<? super T>> List<Interval<T>> merge(List<Interval<T
        List<Interval<T>> sorted = new ArrayList<>(xs);
        sorted.sort(Comparator.comparing(Interval::lo));
        List<Interval<T>> out = new ArrayList<>();
        for (Interval<T> iv : sorted) {
            if (out.isEmpty() || iv.lo().compareTo(out.get(out.size() - 1).hi()) > 0
                out.add(iv);
            } else {
                Interval<T> last = out.remove(out.size() - 1);
                T hi = iv.hi().compareTo(last.hi()) > 0 ? iv.hi() : last.hi();
                out.add(new Interval<>(last.lo(), hi));
            }
        }
        return out;
    }

    static <T extends Comparable<? super T>> String show(List<Interval<T>> xs) {
        StringBuilder sb = new StringBuilder();
        for (Interval<T> iv : xs) sb.append(sb.length() > 0 ? " " : "").append(iv);
        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Interval<Integer>> ints = new ArrayList<>();
        for (int i = 0; i < n; i++) ints.add(new Interval<>(in.nextInt(), in.nextInt
        int m = in.nextInt();
        List<Interval<String>> words = new ArrayList<>();
        for (int i = 0; i < m; i++) words.add(new Interval<>(in.next(), in.next()));
        List<Interval<Integer>> mi = merge(ints);
        List<Interval<String>> mw = merge(words);
        System.out.println(show(mi));
        System.out.println(show(mw));
        System.out.println("merged=" + mi.size() + "+" + mw.size());
    }
}
```

CHECKPOINT QUESTIONS

1. The main benefit of `List<String>` over raw `List` is…

A. Speed

B. Compile-time checking of element types and no casts on reads

C. Smaller memory

D. Thread safety

2. `static <T> T pick(List<T> xs)` — `T` is inferred from…

A. The method body

B. The argument's type (and the target type if needed)

C. Always `Object`

D. A required explicit witness

3. `class Box<T> { T[] arr = new T[4]; }` fails with…

A. Nothing

B. "generic array creation" — arrays need a run-time element type erasure removed

C. "missing bound"

D. "T is abstract"

4. `<T extends Comparable<T>>` versus `<T extends Comparable<? super T>>`: the second additionally accepts…

A. Primitives

B. Types whose `Comparable` implementation comes from a superclass

C. Unrelated types

D. Nulls

5. A method parameter you only iterate and read should be typed…

A. `List<T>`

B. `List<? extends T>`

C. `List<? super T>`

D. `List<Object>`

6. A method parameter you only add `T`s to should be typed…

A. `List<? extends T>`

B. `List<? super T>`

C. `List<?>`

D. `List<Object>`

7. Which is legal?

A. `List<String>[] arr = new List<String>[3];`

B. `List<?>[] arr = new List<?>[3];`

C. `T t = new T();`

D. `catch (MyGenericException<T> e)`

8. Overloading `f(List<String>)` and `f(List<Integer>)` in one class…

A. Works

B. Fails: both erase to `f(List)` — a name clash

C. Works with `@Override`

D. Works if one is static

9. `(List<String>) obj` compiles with…

A. No warning

B. An unchecked warning — only the raw `List` part is checked at run time

C. A compile error

D. A runtime check of every element

10. Bridge methods are visible…

A. In source code

B. In stack traces and reflection (`Method.isBridge()`)

C. Nowhere

D. Only in interfaces

11. `Class<T>` tokens are used to…

A. Speed up generics

B. Recover type information at run time for instantiation, arrays and checked casts

C. Avoid bounds

D. Replace wildcards

12. `static <T> void swap(T[] a, int i, int j)` works with `Integer[]` but not `int[]` because…

A. Arrays cannot be generic

B. `int[]` is not a `T[]` — primitives are not reference types

C. The method must be non-static

D. `swap` is reserved

13. Where should wildcards generally **not** appear?

A. API parameters

B. Return types and local variables

C. Producer parameters

D. Consumer parameters

14. A record can be generic: `record Pair<A, B>(A first, B second) {}`.

A. False

B. True — records support type parameters like classes

C. Only with one parameter

D. Only if final

ANSWER KEY · CHECKPOINT — GENERICS

1. B Errors move from a distant `ClassCastException` to the line that inserted the wrong type.

2. B `pick(List.of("a"))` infers `T = String`; a witness `Main.<String>pick(...)` is rarely necessary.

3. B Use `Object[]` with a contained cast, or `Array.newInstance` with a `Class<T>`.

4. B The wildcard lets `Manager` qualify through `Employee implements Comparable<Employee>`.

5. B Producer → `extends`. It accepts `List<SubtypeOfT>` too.

6. B Consumer → `super`. It accepts `List<SupertypeOfT>` too.

7. B `List<?>` is reifiable, so an array of it may be created; the others are forbidden by erasure.

8. B Erasure leaves identical signatures.

9. B The type argument cannot be verified; a wrong promise surfaces later as `ClassCastException`.

10. B They are synthetic, generated to keep overriding correct under erasure.

11. B `Enum.valueOf(Day.class, s)`, `EnumMap`, JSON parsers all take one.

12. B Primitive arrays need their own overloads; `int[]` cannot be a `T[]` for any `T`.

13. B A wildcard return forces every caller to handle it; locals should be concrete.

14. B Generic records are the idiomatic small value carriers.