🚀 Java Interview Prep Series

Ajit Gupta Follow 21 min read · Jun 23, 2025

1 🔍 What's the difference between and ?

Here's the quick breakdown 👇
✅ keyword
• Belongs to the class, not the instance
• Used for shared variables and methods
• Common in utility classes and constants
• No need to create an object to access static members

```
{
```

```
                    { } //
```

🔐 Using   together?
• Used to create constants that belong to the class and never change
• Common in config values, math constants, etc.

```
                                  =  .        ;

                               .  .  .
```

2. 🔍 Polymorphism in Java – Compile-Time vs Run-Time

Let's break it down with clear examples 👇
✅ What is Polymorphism?
Polymorphism means "many forms." In Java, it allows objects to behave differently based on their actual type — even if accessed through a common interface or superclass.
1️⃣ Compile-Time Polymorphism (Method Overloading)
• Same method name, different parameter list
• Resolved by the compiler at compile time
• Used to increase code readability and reusability

```
                  {
       (      ,      ) {
         +  ;
}

          (          ,          ) {
         +  ;
}
}
```

✅ Output:

```
                =                  ();
 .    ( ,  ); //
```

```
   .      (  .  ,   .  ); //
```

2️⃣ Run-Time Polymorphism (Method Overriding)
• Involves inheritance
• Subclass overrides the method of superclass
• Resolved at runtime using dynamic method dispatch

```
                    {
                () {
          .     .            ("                                    ");
}
}


                                       {
                () {
          .     .            ("                    ");
}
}
```

✅ Output:

```
             =            ();
   .        (); //          :
```

🔥 Pro Tip:
Polymorphism enables flexibility and scalability in code — it's the heart of dynamic behavior in OOP.

. . .

3. 🔍 Abstract Class vs Interface in Java

Let's clear the confusion with real insights + examples 👇
✅ What is an Abstract Class?
An abstract class is a class that cannot be instantiated and may contain abstract methods (without a body) as well as concrete methods (with a body).

```
                                 {
                                    ();

                     () {
             .      .          ("                    ...");
}
}
```

✅ What is an Interface?
An interface is a contract that defines what a class can do, without specifying how. All methods were abstract by default (until Java 8+ introduced default and static methods).

```
                              {
                ();  //
}
```

💡 When to Use What?
✅ Use Abstract Class when:
• You want to share code among related classes.
• You need constructors or state.
• You expect the class to evolve in the future with new methods.
✅ Use Interface when:
• You want to define a capability (e.g., Comparable, Serializable).
• You need to support multiple inheritance.
• You're designing APIs for plug-and-play behavior.

👇 Example in Action

```
                                 {


                                    ();
```

```
            () {
        .     .           ("          ...");
}
}

                         {
          ();
}

                                                              {

                   () {
        .     .           ("      !");
}

                     () {
        .     .           ("                        ");
}
}
```

🔥 Pro Tip:
Use abstract classes when you need a shared base structure; use interfaces when you want to define capabilities that can be added to any class.

. . .

4. 🔍 Method Hiding vs Method Overriding in Java

Let's decode it with a crisp example and clear answers 👇
📦 Code Example:

```
                         {
                     () {
        .     .           ("                    ()");
```

```
}
}

                                              {
                  () {
        .   .        ("               ()");
}
}

                                {
                            (        []       ) {
                  =                       ();
    .     ();
}
}
```

📌 Questions & Answers:
1️⃣ What will be the output?
✅ Output: ()
👉 Because static methods are resolved at compile-time using the reference type (   ), not the actual object.
2️⃣ What if   () methods were non-static?
✅ Output: ()
👉 Now it would be a case of method overriding, and Java uses dynamic method dispatch, so the method of the actual object (   ) is called.
3️⃣ What is Method Hiding in Java?
🧠 Method Hiding occurs when:
A static method in a subclass has the same signature as a static method in the parent class.
This does not override it but hides it.
Method resolution depends on the reference type, not the object.

🔥 Pro Tip:
If you're overriding, make methods non-static. If you keep them static, you're

entering method hiding territory, which behaves differently!

. . .

5. 🎯 Many developers mix up the access scope of , , and modifiers. It’s a small concept, but hugely important, so it’s worth preparing well!

🧭 The Four Access Levels:
👉
• Accessible only within the same class
• Most restrictive → Ideal for encapsulation
👉 (no modifier)
• Accessible within the same package
• No external access from other packages
👉
• Accessible within the same package + subclasses outside the package
• Useful in inheritance hierarchy
👉
• Accessible from anywhere
• Least restrictive → Ideal for public APIs

💡 Interview Tip:
• Know what can access what, especially between packages and subclasses.
• Don’t assume is more restrictive than — it’s not in some contexts!

. . .

6. Two interfaces with same variable names but different values — what happens when you implement both?

This is a frequently asked coding problem that checks your grasp on Java interfaces and ambiguity resolution.

🧪 Problem:
What will be the output of the following code?

```
                 {
              = 10;
}

                 {
              = 20;
}

                                   ,   {

                       () {
          .    .            (         );
}

                                   (        []       ) {
         =             ();
  .                ();
}
}
```

✅ Output and Explanation:
In Java, interface variables are implicitly public static final.
Since both interfaces A and B declare   , referring to   alone causes ambiguity → compilation error.

To resolve this, we use qualified names like .   or .   .

🔥 Interview Tip:
• Always remember interface variables are and .
• If two interfaces have same-named constants, use
. to disambiguate.

. . .

⚠️ What is an Exception?

An Exception is an event that disrupts the normal flow of a program. It's a runtime error that can occur due to:
• Invalid user input
• File not found
• Network failure
• Division by zero
...and many more!

🧬 Java Exception Hierarchy:
. .
👉 (Not meant to be caught)
,
👉 (Can be handled)
— must be declared or handled
→ ,
— runtime issues
→ ,

🧑‍💻 Custom Exception in Java:
You can create your own exception class by extending (checked) or (unchecked).

```
                                              {
                      (              ) {
        (    );
```

```
}
}

                {
                                    (         []        )                          {
                                  ("                        !");
}
}
```

· · ·

7. ⚠️ Why Handle Exceptions?

Proper exception handling ensures your app doesn't crash unexpectedly and allows you to:
• Show user-friendly error messages
• Perform cleanup tasks (like closing files/DB connections)
• Avoid leaving your system in an inconsistent state

🔁 Syntax Breakdown:

```
     {

//

}          (                        ) {

//

}             {

//                                    (                                )

}
```

🧑‍💻 Example:

```
                                              {

                                  (          []        ) {

      {

                =     /   ;
                             (“          : “ +          );

}          (                                ) {

                             (“                        : “ +                   ());

}             {

                             (“                                 ”);

}

}

}
```

🧠 Output:

```
                    : /
```

✅ Key Notes:
• You can’t skip the catch or finally block together. One must exist.
• The finally block always executes, even if an exception is thrown or a return is called in try/catch.
• You can have multiple catch blocks for different exception types.

⚡ Common Mistakes:
• Catching a generic Exception too early in the hierarchy.
• Forgetting resource cleanup (unless using try-with-resources).

. . .

1️⃣ **vs**
2️⃣ **vs** **vs** **()**
We often get confused with these keywords due to similar spelling but very different purposes — so better to prepare them well!

✅ vs
👉 :
• Used to explicitly throw an exception from a method or block.
• Used inside method body.

```
                                    ("                      ");
```

👉 :
• Declares the possibility of exceptions for a method.
• Written in the method signature.

```
                  ()                         ,                  {     }
```

✅ vs vs ()
👉 :
• Used to make variables immutable, methods un-overridable, and classes unextendable.

```
              =   ;
```

👉 :
• A block that always executes after try-catch regardless of exception occurrence.

```
      {       }          (                    ){       }              {       }
```

👉 ():
• A method in Object class called by the Garbage Collector before destroying an object.
• Rarely used now; deprecated in recent Java versions.

```
@
                                          ()                              {

           .     .            (“                                            ”);

}
```

. . .

Multithreading in Java.
Whether it's improving performance, managing tasks in parallel, or building responsive apps, multithreading is an essential skill for any Java developer.

🧠 What is Multithreading?
Multithreading is a programming technique where multiple threads run concurrently, sharing the same process memory.
Each thread is an independent path of execution — ideal for multitasking!

💡 Why use Multithreading?
• Faster execution
• Efficient CPU usage
• Better resource management
• Useful in I/O operations, games, UI, etc.

🛠️ How to Create Threads in Java?
👉 By extending Thread class

```
                                                {
                        () {
          .    .          ("                                           ");
}
}
                            {
                                    (         []       ) {
              1 =                         ();
  1.        ();
}
}
```

👉By implementing Runnable interface

```
                                                          {
                        () {
          .    .          ("
              ");
}
}
                            {
                                    (         []       ) {
            1 =                    (                    ());
  1.        ();
}
}
```

🤔 Which approach to choose?

Use Runnable if:

• Your class needs to extend another class

• You want to separate task (Runnable) from thread execution (Thread)

. . .

🔍 , , and in Java

These are important topics in both multithreading and null-safety, and they often appear in interview discussions. Let's break them down 👇

✅ What is ?

• is similar to , but it can return a result and throw a checked exception.

• Used when you want your thread to return a computed value.

```
                                          <         > {
                    ()                   {
       "                     !";
}
}
```

✅ What is ?

• is used to store the result of an asynchronous computation.

• It provides methods to check if the task is complete, wait for completion, and retrieve the result.

```
                           =
          .                         ();
       <       >      =      .    (               ());
       .   .       (     .    ());
```

✅ What is ?

• is a container object used to contain not-null objects.

• It helps avoid NullPointerExceptions.

```
         <       >              =
         .         ("     ");
  (              .          ()) {
          .     .          (                 .     ());
}
```

✨ Why is Powerful:

- () — Run logic only if value exists
- () / () — Provide safe defaults
- () — Cleanly throw exceptions when value is absent
- () / () — Transform data functionally and elegantly

💡 When to Use What?
• Use and for concurrent tasks that return results.
• Use when you want to handle absence of values safely.

. . .

ExecutorService.
It's a powerful feature from the java.util.concurrent package that helps manage thread execution efficiently and cleanly.

🧠 **What is ExecutorService?**
• It's an interface designed to handle asynchronous task execution, letting you focus more on what to run and less on how to run it.
• It abstracts away thread creation and management, giving us easy-to-use tools for executing tasks concurrently.

🔧 Types of ExecutorService:
✅ 1. ()
Executes one task at a time using a single worker thread.

```
                                 =
      .                                        ();
     .          (() ->         .    .         ("
      "));
     .          ();
```

✅ 2. ( )
Executes tasks using a fixed number of threads.

```
                                   =
          .                                 ( );
   (      =  ;   <  ;   ++) {
         .          (() ->          .    .          ("
              "));
}
         .            ();
```

✅ 3. ()
Creates new threads as needed, reuses them when available. Suitable for short-lived async tasks.

```
                                   =
          .                                   ();
   (      =  ;   <  ;   ++) {
         .          (() ->          .    .          ("
              "));
}
         .            ();
```

✅ 4. ( )
Used for delayed or periodic task execution.

```
                                                 =
          .                                       ( );
         .            (() ->          .    .          ("
              "),   ,                        );
         .            ();
```

📌 Quick Recommendations:
• Use () to close the executor gracefully.
• Use for limited resources.

• Use for flexible scaling.
• Use for scheduled tasks.

. . .

Java Memory Model (JMM) — one of the most asked topics in Java multithreading interviews.

🔍 What is JMM?
The Java Memory Model (JMM) defines how threads interact through memory and what behaviors are allowed in concurrent execution. It's a crucial abstraction that ensures visibility, ordering, and atomicity of variables between threads.

💡 Key Concepts:
• Shared Memory: Threads don't communicate by passing messages but by reading/writing shared variables.
• Thread Working Memory: Each thread has its own copy of variables (cached).
• Main Memory: The actual memory shared by all threads.

🔐 Visibility Problem:
Changes made by one thread may not be visible to others due to thread-local caching.
✅ Solution: Use , , or other concurrency utilities.

🚦 Happens-Before Relationship (JMM's backbone):
• A . () call happens-before the first action in the thread
• . () happens-before another thread's . ()
• writes happen-before reads

🧪 Sample Code using :

```
                                             {
                                    =          ;
                         () {
            (              ) {
//
}
}
                          () {
            =           ;
}
}
```

🧵 One thread runs the task, another calls () — due to , changes are visible immediately.

📌 Mastering the JMM helps prevent subtle concurrency bugs and boosts confidence in handling real-world multithreaded applications!

. . .

Cloneable Interface (Marker Interface) & == vs .equals() in Java!

🔍Cloneable Interface:
• Cloneable is a marker interface (contains no methods).
• It tells the JVM that your class supports the clone() method.
• Used to create object copies (shallow by default).

💻 Code implementation:

```
                                              {
          =    ;
                         ()                                       {
          (        )       .        ();
}
}
```

⚠️ If a class doesn’t implement Cloneable and clone() is called, it throws CloneNotSupportedException.

🔍 == vs .equals()
• == → Compares references (memory address).
• .equals() → Compares values (actual content of the object).

💻 Code implementation:

```
              = “        ”;
              =                  (“        ”);


    ==    ; // false
   .          (    ); // true
```

🎯 Interviewers often check if you understand this difference deeply, especially for String, Integer, and custom objects.

· · ·

🔍 Shallow Copy vs Deep Copy in Java — What’s the difference?

🔹 Shallow Copy:
• Copies the object and references to the nested objects.
• Changes in the nested object affect both copies.
• Performed using clone() (default behavior).

🔸 Deep Copy:
• Copies the object and creates new instances of nested objects.
• Original and copy are completely independent.
• Requires manual handling or serialization.

💻 Code Example: Shallow Copy

```
                 {
              ;
         (              ) {      .     =      ; }
}

                                          {
              ;
               ;
        (             ,               ) {
     .     =      ;
     .     =      ;
}
                        ()                                        {
         (        )       .      ();
}
}
```

💻 Code Example: Deep Copy

```
                        ()                                        {
                 = (        )       .      ();
        .     =             (     .     .     );
                ;
}
```

🎯 Interview Insight:
• Always clarify whether the interviewer is referring to shallow or deep cloning.
• Demonstrating both approaches shows depth in understanding object memory management.

. . .

🧠 **What is Garbage Collection?**
Garbage Collection in Java is the process of automatically identifying and reclaiming memory occupied by objects no longer in use, preventing memory leaks and improving performance.

🔄 Java Memory Areas Relevant to GC
• Young Generation (Eden + Survivor spaces) — where new objects are created
• Old Generation (Tenured space) — where long-lived objects go
• Permanent Generation / Metaspace (Java 8+) — for class metadata

🔍 Types of Garbage Collectors
1️⃣ Serial GC (Single-threaded)
Best for small apps with limited memory.
Use with: -   :+
2️⃣ Parallel GC (Throughput collector)
Uses multiple threads for GC in young and old gen.
Use with: -   :+
3️⃣ CMS (Concurrent Mark-Sweep)
Minimizes pause time by doing most GC work concurrently.
Use with: -   :+
4️⃣ G1 GC (Garbage First)
Breaks heap into regions and performs parallel + concurrent collection (Default from Java 9+).
Use with: -   :+
5️⃣ ZGC & Shenandoah (Low latency collectors)
Designed for massive heaps and ultra-low pause times.
Use with: -   :+
-   :+

🧰 GC Algorithms (At a glance):
👉Mark and Sweep — Mark live objects, sweep the rest.
👉Copying — Copies live objects to a new space.

👉Generational — Based on object age (Young vs Old gen).
👉Region-based (G1, ZGC) — Breaks heap into regions for efficient GC.

✅ GC tuning is key for performance-critical apps, especially when memory footprint and response times matter!

. . .

Java 8 Features you must know for interviews:

1️⃣ Lambda Expressions
Enables treating functionality as a method argument or passing behavior.

```
      <         >        =        .        ("           ", "      ",
"        ");
      .          (      ->        .   .         (      ));
```

2️⃣ Functional Interfaces
Interfaces with a single abstract method, used as the basis for lambda expressions.
Examples: ,   ,   < >,   < >

3️⃣ Stream API
Provides a functional approach to processing collections.

```
      <         >          =        .         ()
.         (       ->      .             (" "))
.          (             .        ());
```

4️⃣ Default Methods in Interfaces
Interfaces can now have method implementations.

```
                             {
                      () {
```

```
          .     .           ("                      ");
}
```

5️⃣ Method References

Simplified syntax for calling methods via "::".

```
        .           (         .    ::         );
```

6️⃣ Optional Class

To avoid NullPointerException and handle absence of value gracefully.

```
            <          >      =            .               ("         ");
     .              (         .    ::         );
```

7️⃣ New Date and Time API (java.time)

Immutable and more human-friendly replacements for Date and Calendar.

```
                      =              .     ();
                     =              .     ();
```

8️⃣ Collectors Class

Used with Stream API to collect elements into collections or summaries.

```
    <          ,      <         >>                    =
        .        ().         (              .             (         ::
  ));
```

9️⃣ Parallel Streams

Makes it easy to process collections in parallel for performance gains.

```
        .                   ().         (        .    ::          );
```

. . .

🔧 **Internal Working of HashMap:**

A HashMap stores key-value pairs. Internally, it uses an array of buckets, where each bucket is a LinkedList (pre-Java 8) or a tree structure (post-Java 8, in case of high collisions).

♻️ Here's how the process works:
1️⃣ () is called on the key.
A hash function applies a transformation:
= (= ()) ^ (>>>)
2️⃣ The index in the array is calculated using:
= & (—)
3️⃣ If a bucket is empty, the entry is stored.
4️⃣ If a bucket is occupied, it checks for key equality:
5️⃣ If keys are equal, it updates the value.
If not, a collision occurs, and the new entry is added to the chain (LinkedList or Tree).

💻 Java 8 Collision Change:
👉 Before Java 8:
• Collisions were handled via LinkedList chaining.
• O(n) time complexity for search in case of many collisions.
👉 After Java 8:
• When the number of entries in a bucket exceeds _
(default: 8), the list is converted into a balanced Red-Black Tree.
• Reduces search time from O(n) to O(log n) in high-collision scenarios.
🔄 Treeification only happens if the underlying array size is greater than
_ _ (default: 64).

🧠 Quick Tip for Interviews:
• Be ready to explain why HashMap was improved (performance degradation due to many collisions).
• Know the constants:

```
          _                    =
              _                    =
      _               _                 =
```

. . .

🔍 ArrayList vs LinkedList: Key Differences

✅ ArrayList:
• Uses a dynamic array as its underlying data structure.
• Has lower memory usage (stores only elements).
• Provides fast random access — ( ) time to get elements by index.
• Insertions and deletions (except at the end) are slower — ( ) time due to shifting elements.
• Cache friendly because elements are stored in contiguous memory.
✅ LinkedList:
• Uses a doubly linked list as its underlying data structure.
• Has higher memory usage (stores elements plus node pointers).
• Random access is slow — ( ) time as it must traverse nodes.
• Insertions and deletions (especially at the beginning or middle) are fast — ( ) time by updating links.
• Less cache friendly since nodes are scattered in memory.

♻️ When to Use ArrayList?
• Frequent read operations (searching, accessing by index)
• Memory efficiency matters
• Less frequent insertions/deletions in the middle of the list
♻️ When to Use LinkedList?
• Frequent insertions/deletions (especially at the beginning/middle)
• Queue-like operations (add/remove from both ends)
• Memory overhead is acceptable

👇 Example Usage

```
        .      . ;

      <         >          =                    <>();
  .     (“       ”);
  .     (  ); //              !

        <         >          =                    <>();
    .         (“          ”);
    .               (); //         /                         !
```

💡 Pro Tip:
• Choose ArrayList for fast random access and low memory use.
• Choose LinkedList for frequent insertions/deletions or when you need queue operations.

. . .

What’s the difference between Comparable and Comparator, and when should you use each? Let’s dive in! 👇

✅ Comparable Interface:
• It is part of the java.lang package.
• Used to define the natural ordering of objects by implementing the compareTo() method inside the class itself.
• Modifies the class whose objects need to be sorted.
• Supports a single sorting sequence (e.g., sorting by age or name, but only one at a time).
• Used when there is a clear default way to order objects.
• Sorting is done using Collections.sort(list) or Arrays.sort(array) which rely on the class’s compareTo() method.
• Example: A Person class implementing Comparable<Person> to sort by age.

💻 Code example :

```
                                        <         > {
              ;
          ;

@
                  (         ) {
          .       (    .   ,  .   );
}
}
```

✅ Comparator Interface:
• It belongs to the java.util package.
• Defines custom ordering by implementing the compare(Object o1, Object o2) method in a separate class or via lambda expressions.
• Does not require modifying the original class.
• Supports multiple sorting sequences — you can create different comparators to sort by name, age, salary, etc.
• Useful when sorting criteria vary or when you cannot change the class's code.
• Sorting is done via Collections.sort(list, comparator) or list.sort(comparator).
• Java 8 enhanced Comparator with methods like comparing(), thenComparing(), reversed(), and null handling (nullsFirst(), nullsLast()).
• Example: A PersonNameComparator class or a lambda expression to sort by name.

💻 Code example :

```
          .    .          ;



          <        > {
@
                (          ,          ) {
          .    .        (  .    );
}
}
```

🔥 Pro Tip:
• Use Comparable when your class has a natural, single way to be ordered.
• Use Comparator when you need flexibility for multiple sorting criteria or cannot modify the class.

. . .

Why does modifying an ArrayList during iteration throw a ConcurrentModificationException?

🔍 The Interview Scenario
Question: “Write code to remove all even numbers from an ArrayList while iterating.”
Many candidates write something like this:

```
      <           >           =                  <>(         .        ( ,  ,
 ,  ,  ));
     (                :          ) {
   (     %   ==  ) {
           .       (    ); // ❌ Throws ConcurrentModificationException!
}
}
```

Result:

```
                       “      ”

      .     .
```

💡 Root Cause Analysis
• Java’s ArrayList uses a fail-fast iterator.
• When you use a for-each loop, it internally uses an Iterator.
• If you structurally modify the list (e.g., add(), remove()) directly on the list (not via the iterator) during iteration, it invalidates the iterator’s state.

• As a result, the iterator throws a ConcurrentModificationException to prevent unpredictable behavior.

✅ The Fix:
1️⃣ Use Iterator.remove()

```
      <            >            =                    <>(         .         ( ,  ,
 ,  ,  ));
            <            >     =             .             ();
        (    .             ()) {
                    =    .        ();
    (      %    ==   ) {
    .          (); // This will help with safe removal!
}
}
```

2️⃣ Use CopyOnWriteArrayList(For thread-safe scenarios)

```
      <            >            =                                   <>
(          .          ( ,  ,  ,  ,  ));
     (                   :            ) {
    (      %    ==   ) {
           .          (      ); // No exception (But it’s inefficient for large,
frequently modified lists, as it creates a new copy on every write)
}
}
```

🔥 Pro Tip
• Single-threaded? Always use Iterator.remove().
• Multi-threaded? Opt for CopyOnWriteArrayList or ConcurrentHashMap.
• Avoid modifying collections directly during iteration.

. . .

🔹 **What is a Functional Interface?**

A functional interface has exactly one abstract method. It can be implemented with a lambda expression or method reference.
Examples: Runnable, Comparator, and all interfaces below.

✅ Predicate

• Takes one input, returns a boolean (true/false).
• Great for filtering!

```
              <      []>              =        -> (      .             ==   );
   (                .        (             []{ }))
          .     .            (“          !”);
```

✅ Function

• Takes one input, returns a result.

```
            <          ,             >                 =   ->   .            ();
          =                 .         (“        ”); //
```

✅ Consumer

• Takes one input, returns nothing.
• Used for actions like printing.

```
            <          >                =   ->
          .     .           (  .                   ());
                .         (“        ”);
```

✅ Supplier

• Takes no input, returns a result.

```
            <          >            = () ->        .           ();
          .     .           (          .    ());
```

✅ BiPredicate

• Takes two inputs, returns a boolean.

```
               <          ,          >          = (  ,  ) ->  .        (  );
(           .        (“       ”, “       ”))          .    .         (“         !”);
```

✅ BiConsumer

• Takes two inputs, returns nothing.

```
             <          ,          >              = (  ,  ) ->
      .    .          (  + “: “ +  );
          .       (“     ”,    );
```

✅ BiFunction

• Takes two inputs, returns a result.

```
             <          ,          ,          >          = (  ,  ) ->  +  ;
      .    .          (          .          (“   , “, “        !”));
```

✅ UnaryOperator

• Takes one input, returns a result of the same type.

```
                 <          >          =  ->  *  ;
      .    .          (          .          (  )); //
```

✅ BinaryOperator

• Takes two inputs of the same type, returns a result of the same type.

```
                  <          >       = (  ,  ) ->  +  ;
      .    .          (    .         (  ,  )); //
```

💡 Pro Tip:

Functional interfaces make your code concise, readable, and powerful — especially with streams and lambdas. Try these out in your next project!

How to create streams in Java 8 and explore the most common methods to do so. Streams are powerful for processing collections of data in a clean and functional style.

✅ 1. Create a Stream from a Collection

Most common way — call .stream() on any Collection like List, Set, or Queue.

```
       <String>          =          .          (“          ”, “          ”,
“               ”);
          <String>                =          .          ();
```

✅ 2. Create a Stream from an Array

Use   .   () to convert arrays into streams.

```
String[]          = {“          ”, “          ”, “               ”};
          <String>                =          .          (          );
```

✅ 3. Create a Stream using Stream.of()

Create a stream from fixed elements or an array.

```
          <String>                =          .   (“          ”, “          ”,
“               ”);
```

✅ 4. Create Infinite Streams with Stream.generate()

Generates an infinite stream using a Supplier.

```
          <Double>                     =
          .            (      ::          ).          (5);
                     .          (          .     ::          );
```

✅ 5. Create Infinite Streams with Stream.iterate()
Generates an infinite stream by applying a function repeatedly.

```
          <Integer>            =          .            (1,   ->   + 1).        (5);
           .           (          .    ::           );
```

✅ 6. Create a Stream from a File (Lines)
Use Files.lines() to get a stream of lines from a file.

```
     (          <String>          =
        .        (        .      (“           .txt”))) {
        .           (         .    ::           );
}         (                   ) {
   .                      ();
}
```

💡 Pro Tip:
Streams don’t store data; they operate on data sources like collections, arrays, or I/O channels. Use intermediate operations (filter, map) and terminal operations (forEach, collect) to process data efficiently.

. . .

Java 8 Stream intermediate operations — these are lazy (run only when a terminal operation is called) and return a new stream.

1️⃣ filter(Predicate<T> predicate)
• Takes: Predicate<T> (returns boolean)
• Use: Keep elements matching a condition

```
      .         ()
.        (  ->   %   ==  )
.          (         .    ::           );
```

2️⃣ map(Function<T, R> mapper)

• Takes: Function<T, R> (T → R)

• Use: Transform each element

```
          .            ()
.      (   ->   .                      ())
.              (                    .            ());
```

3️⃣ flatMap(Function<T, Stream<R>> mapper)

• Takes: Function<T, Stream<R>>

• Use: Flatten nested collections

```
            .            ()
.              (                    ::           )
.              (                    .            ());
```

4️⃣ distinct()

• Takes: nothing

• Use: Remove duplicates

```
        .            ()
.                ()
.              (                    .            ());
```

5️⃣ sorted() / sorted(Comparator<T> comparator)

• Takes: nothing (natural order) or Comparator<T>

• Use: Sort elements

```
            .            ()
.            ()
.              (           .      ::             );
```

6️⃣ peek(Consumer<T> action)

• Takes: Consumer<T> (performs an action, returns nothing)

• Use: Debug/inspect elements

```
      .        ()
.      (  ->         .    .         (  ))
.          (              .        ());
```

7️⃣ limit(long maxSize)
• Takes: long
• Use: Limit number of elements

```
         .       ( )
.          (        .    ::         );
```

8️⃣ skip(long n)
• Takes: long
• Use: Skip first n elements

```
         .     ( )
.          (        .    ::         );
```

💡 Pro Tip:
• Know what argument each method expects to chain streams effectively.
• No terminal operation = no execution!

. . .

Java 8 Stream terminal operations — the final step that produces a result or side effect and ends the stream pipeline.

1️⃣ forEach(Consumer<T> action)
Performs an action for each element (side effect, like printing).

```
      .        ()
.          (        .    ::         );
```

2️⃣ collect(Collector<T, A, R> collector)
Gathers elements into a collection (List, Set, Map, etc).

```
      <           >         =       .         ()
.         (  ->   %    ==  )
.           (                 .         ());
```

3️⃣ reduce(BinaryOperator<T> accumulator)
Combines elements into a single result (sum, concat, etc).

```
          =       .         ()
.         ( , (  ,   ) ->   +  );
```

4️⃣ count()
Returns the number of elements.

```
              =       .         ().       ();
```

5️⃣ min(Comparator<T> comparator) / max(Comparator<T> comparator)
Finds the minimum or maximum element (returns Optional).

```
            <          >      =      .        ().    (          ::          );
```

6️⃣ anyMatch(Predicate<T> predicate), allMatch, noneMatch
Returns boolean if any, all, or none elements match a condition.

```
                      =      .        ().          (  ->   %   ==  );
```

7️⃣ findFirst() / findAny()
Returns the first or any element (as Optional).

```
            <          >        =      .        ().          ();
```

💡 Pro Tip:
Terminal operations trigger the pipeline — once called, the stream is consumed and can’t be reused.

. . .

Java 8 Parallel Streams — a simple way to process large or CPU-intensive data faster by leveraging multiple CPU cores. But when should you use them, and what are the caveats?

👉 What is a Parallel Stream?
A parallel stream splits your data into chunks and processes them concurrently across multiple threads, utilizing all available CPU cores for faster results — especially useful for big data or heavy computations.

👉 How to Create a Parallel Stream?
1️⃣ From a collection:

```
      .                      ()
.         (  ->   %    ==  )
.           (         .    ::          );
```

2️⃣ From any stream:

```
         .            ()
.    (  ->      )
.           (         .    ::          );
```

3️⃣ Example with numbers:

```
              .                 ( ,   )
.           ()
.           (  ->         .    .         ("         : " +
         .                   ().        () + ",       : " +  ));
```

👉 When Should You Use Parallel Streams?
• For large data sets or CPU-bound tasks (e.g., heavy calculations, data transformations).
• When operations are stateless and independent (no shared mutable state).
• On multi-core machines for maximum benefit.

👉 When Should You Avoid Parallel Streams?
• For small data sets (thread overhead > speedup).
• For I/O-bound operations (disk/network reads).
• If your code relies on element order or uses shared mutable state.
• When you need deterministic processing order (parallel streams may process out of order).

♻️ Performance Tips
• Always benchmark before and after using parallel streams — sometimes sequential is faster for small or simple tasks.
• Arrays of primitives split and process most efficiently in parallel.
• Avoid parallel streams for tasks that require synchronization or have expensive merge operations.

💡 Pro Tip:
• Parallel streams are powerful for the right use case, but not a magic bullet.
• Use them for big, CPU-heavy, stateless tasks — test and measure before making the switch!

4

Written by Ajit Gupta
165 followers · 208 following

Follow

No responses yet

Write a response

What are your thoughts?

More from Ajit Gupta

Ajit Gupta

🚀 Java Microservices Interview Cheat Sheet (20 Q&A)

Jun 16 46

Ajit Gupta

🚀 Interview Question : Java & Spring Boot Insights🌟 ✅ Q.

Jun 23 10 2

Ajit Gupta

All Oracle Senior Java Developer Interview Questions

Ajit Gupta

☕ Java & Spring Interview Questions for Experienced…

🚀 Microservices with Spring Boot — Step-by-Step Beginner Guide

Jun 13 11

1d ago 16

See all from Ajit Gupta

Recommended from Medium

In Coding Odyssey by Shivam Srivastava

Qualcomm Java Interview Experience—2

Interview of Professional with 6+ Years of Experience

5d ago 13

In Stackademic by Kavya's Programming Path

I Reviewed 500 Pull Requests—Here's What Every Java Dev Gets...

From Optional disasters to stream overkill, these are the Java sins haunting your code...

6d ago 131 6

PraveenCodes

Full Stack With Ram

JPMorgan Chase & Co. Interview Experience

Java Backend Developer — JPMorgan Chase & Co. Interview Experience (5 Years)...

Jul 8 · 7 · 1

Top 20 Java 8 Streams Coding Interview Questions Every...

Java 8 brought a paradigm shift to how we write code, and one of its most powerful...

Jul 3 · 1

In Javarevisited by Pudari Madhavi

Cracking the Barclays Java Developer Interview: A Step-by-...

If you are not a Member — Read for free here :

Jun 2 · 20

Ajit Gupta

Java Developer Interview

Hey everyone!

May 20

See more recommendations