`ClassNotFoundException` and `NoClassDefFoundError` are both **Java exceptions related to missing classes**, but they occur in **different contexts** and have different causes.

🔍 Key Differences

Feature	`ClassNotFoundException`	`NoClassDefFoundError`
Type	Checked Exception (must be handled or declared)	Error (unchecked, usually fatal)
When It Occurs	At **runtime**, when trying to **dynamically load** a class	At **runtime**, when the JVM **can't find a class** it previously compiled
Common Causes	- Using `Class.forName("SomeClass")`- Reflection	- Missing `.class` file at runtime- Classpath issue
Compilation	Class not required at compile time	Class **was present at compile time**, but missing later
Recoverable?	Yes, since it's a checked exception	Usually not — serious error

---

📓 `ClassNotFoundException` Example

Occurs typically during **reflection or dynamic class loading**:

```
try {
    Class.forName("com.example.MyClass");
} catch (ClassNotFoundException e) {
    e.printStackTrace(); // Class not found at runtime
}
```

- This happens when the class name provided doesn't exist in the classpath at runtime.
- This must be **caught or declared** in the method signature.

📕 `NoClassDefFoundError` Example

Occurs when a class was **present at compile time** but **missing at runtime**:

```
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // If MyClass.class is missing at runtime, you'll
get NoClassDefFoundError
    }
}
```

Even though it compiled successfully, if `MyClass.class` is not found at runtime, you'll get:

```
Exception in thread "main" java.lang.NoClassDefFoundError: MyClass
```

🎯 Summary

Situation	Use
Trying to load a class via reflection or `Class.forName()` and it doesn't exist →	`ClassNotFoundException`
Class was available at **compile time** but missing or corrupted at **runtime** →	`NoClassDefFoundError`

In Java, both **Errors** and **Exceptions** are subclasses of the class `Throwable`, but they represent very different kinds of problems:

⚖️ Key Differences: Error vs Exception

Aspect	Error	Exception
Definition	Serious issues from which recovery is **not expected**	Conditions that a program **should handle or recover from**
Type	Subclass of `java.lang.Error`	Subclass of `java.lang.Exception`
Recoverable?	Usually **not recoverable**	Often **recoverable** with proper handling
Examples	`OutOfMemoryError`, `StackOverflowError`, `NoClassDefFoundError`	`IOException`, `NullPointerException`, `SQLException`
Handled via try-catch?	No (not recommended or useful)	Yes, using `try-catch-finally` blocks
Use Case	Indicates JVM or system-level failure	Indicates application-level issues

---

☐ Class Hierarchy

```
Throwable
├── Error
│   └── (e.g. OutOfMemoryError, StackOverflowError)
└── Exception
    ├── Checked Exception (e.g. IOException, SQLException)
    └── Unchecked Exception (e.g. NullPointerException, ArithmeticException)
```

☐ Error Example

```
public class Main {
    public static void main(String[] args) {
        while (true) {
            new Main(); // Eventually causes OutOfMemoryError
        }
    }
}
```

Output:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

📚 Exception Example

```
try {
    int x = 5 / 0; // Causes ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("You can't divide by zero!");
}
```

✓ Summary

Question	Error	Exception
Part of normal app flow?	✗ No	✓ Yes
Should you handle it?	✗ Rarely	✓ Almost always
User-defined?	✗ Not typical	✓ Common and encouraged

In Java, `throw`, `throws`, and `Throwable` are **related to exception handling**, but they serve very different purposes. Here's a clear breakdown:

◆ `throw` – Used to *actually throw* an exception

- Used inside a method or block to throw an exception explicitly.
- You can throw **only one** exception at a time.
- It must throw an object of type `Throwable` (usually `Exception` or its subclass).

✅ Example:

```
throw new IllegalArgumentException("Invalid input!");
```

◆ `throws` – Used in a method signature to *declare* exceptions

- Used in method declarations to indicate **which checked exceptions** the method might throw.
- Can declare **multiple exceptions** separated by commas.
- Helps callers know they must handle or propagate those exceptions.

✅ Example:

```
public void readFile() throws IOException, FileNotFoundException {
    // code that may throw these exceptions
}
```

◆ `Throwable` – The superclass of all errors and exceptions

- `java.lang.Throwable` is the **base class** for anything that can be thrown or caught.
- It has two direct subclasses:
  - `Error` (serious problems, not meant to be caught)
  - `Exception` (recoverable problems)

✅ Example:

```
public void log(Throwable t) {
    System.out.println("Caught throwable: " + t.getMessage());
}
```

🔁 Summary Table

Keyword/Class	Type	Purpose	Example
`throw`	Keyword	Used to **throw** an exception manually	`throw new NullPointerException();`
`throws`	Keyword	Used to **declare** exceptions in method signature	`public void method() throws IOException {}`
`Throwable`	Class	Superclass of **all exceptions and errors**	`catch (Throwable t)`

☐ Quick Tips

- Use `throw` **inside** a method or block.
- Use `throws` in the **method declaration**.
- Catch `Throwable` only when absolutely necessary (e.g., logging all errors/exceptions).

🔗 Chained Exceptions in Java

Chained exceptions in Java allow you to associate one exception with another. This is helpful when:

- One exception causes another.
- You want to preserve the original error while throwing a higher-level or more meaningful exception.

✓ Why Use Chained Exceptions?

To **track the root cause** of an exception through layers of abstraction.
For example:
A `DatabaseConnectionException` might wrap a `SQLException` that was the real problem.

🔧 Syntax for Chained Exceptions

1. Using Constructor

```
Throwable cause = new NullPointerException("Root cause");
Exception e = new Exception("Higher-level exception", cause);
throw e;
```

2. Using `initCause()`

```
Exception e = new Exception("Wrapper exception");
e.initCause(new IllegalArgumentException("Underlying cause"));
throw e;
```

⚠ You can only call `initCause()` once.

🔍 How to Access the Cause

```
try {
    throw new Exception("Top level", new IOException("File not found"));
} catch (Exception e) {
    System.out.println("Caught: " + e);
    System.out.println("Cause: " + e.getCause());
}
```

Output:

```
Caught: java.lang.Exception: Top level
Cause: java.io.IOException: File not found
```

🏗 Custom Exception with Chaining

```
class MyException extends Exception {
    public MyException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

☐ Best Practices

- Always preserve the original cause when re-throwing exceptions.
- Don't lose valuable stack trace info by swallowing exceptions.
- Use `getCause()` to trace the root of complex error chains.

You've compiled a **thorough and excellent list of Java constructor rules**, covering key principles and common pitfalls. Here's a **cleaned-up and structured version**, which you can use for **study, documentation, or teaching purposes**.

✅ Basic Rules for Constructors in Java

---

1 Constructor Name Must Match Class Name

- A constructor must have **exactly the same name as the class**.
- If not, it is considered a method (especially if it has a return type).

```
class A {
    A() {
        // ✅ Valid constructor
    }

    // ❌ This is a method, not a constructor
    void A1() {
    }
}
```

---

2 Constructors Must Not Have a Return Type

- If a return type (even `void`) is specified, it's treated as a method, **not a constructor**.

```
class A {
    A() {
        // ✅ Valid constructor
    }

    void A() {
        // ❌ Treated as a method, not a constructor
    }
}
```

---

3 Every Class Has At Least One Constructor

- If no constructor is defined, Java provides a **default constructor**:

```
class A {
    // No explicit constructor
}

// Compiler adds:
class A {
    public A() {
        // default constructor
    }
}
```

---

4 Constructors Can Be Private

- Useful for **singleton pattern** or **factory methods**.

```
class A {
    private A() {
        // Private constructor
    }

    void insideClass() {
        A a = new A(); // ✅ Allowed
    }
}

class Main {
    public static void main(String[] args) {
        // A a = new A(); ❌ Not allowed (private constructor)
    }
}
```

---

5 Constructor Overloading is Allowed

- A class can have **multiple constructors with different parameters**.

```
class A {
    A() {}
    A(int x) {}
    A(int x, int y) {}
}
A a1 = new A();          // Uses no-arg constructor
A a2 = new A(10);        // Uses one-arg constructor
A a3 = new A(10, 20);    // Uses two-arg constructor
```

---

6 Duplicate Constructors Are Not Allowed

- Constructors must differ in their parameter **types or count**.

```
class A {
    A(int x) {}
    // A(int x) {} ❌ Compile-time error: duplicate constructor
}
```

---

7 Duplicate Parameter Names Not Allowed

- You cannot declare two parameters with the same name.

```
class A {
    // A(int x, int x) {} ❌ Compile-time error
}
```

---

8 Only `public`, `protected`, `private` Are Allowed as Modifiers

- You **cannot** declare constructors as `final`, `static`, or `abstract`.

```
class A {
    // final A() {} ❌ Not allowed
    // static A() {} ❌ Not allowed
    // abstract A() {} ❌ Not allowed
}
```

---

9 First Statement in a Constructor Must Be `super()` or `this()`

- If not explicitly stated, Java inserts `super()` by default.
- `this()` calls another constructor in the **same class**.
- You can only call `super()` or `this()` as the **first statement**.

```
class A {
    A() {
        System.out.println("No-arg constructor"); // super() is inserted automatically
    }

    A(int x) {
        this(); // ✅ Calls no-arg constructor
        System.out.println("One-arg constructor");
    }

    A(int x, int y) {
        super(); // ✅ Explicit call to superclass constructor
        System.out.println("Two-arg constructor");
    }
}
class A {
    A(int x, int y, int z) {
        System.out.println("Invalid if super() or this() follows this line");
        // super(); ❌ Compile-time error
    }
}
```

🔁 10 Recursive or Cyclic Constructor Calls Are Not Allowed

❌ Recursive

```
class A {
    A() {
        this(); // ❌ Constructor calls itself → compile-time error
    }
}
```

❌ *Cyclic*

```
class A {
    A() {
        this(10); // Calls A(int)
    }

    A(int x) {
        this(); // Calls A(), forms a loop → compile-time error
    }
}
```

☐ Summary Table

Rule	Description
✓ Constructor name = class name	Must match exactly
✓ No return type	Adding one turns it into a method
✓ Default constructor	Auto-generated if none exists
✓ Constructor overloading	Multiple constructors allowed
✗ No duplicate signatures	Constructor params must differ
✗ No duplicate parameter names	Param names must be unique
✗ Modifiers like `static`/`final`/`abstract`	Not allowed
✓ First line = `super()` or `this()`	Must be first statement
✗ No recursive or cyclic calls	Causes compile error
✓ Private constructor	For restricted instantiation

That's an **excellent and detailed walkthrough** of how **Java source files are named, compiled, and executed** when they contain **multiple class definitions**. Here’s a **refined, structured summary** of those rules for easy learning and quick reference:

✅ Java File Naming, Compilation & Execution Rules (Multiple Classes)

🔹 Rule #1: File Can Contain Multiple Classes

✅ *Valid Example:*

```
class ClassOne {
    public static void main(String[] args) {
        ClassTwo.methodOfClassTwo();
    }
}

class ClassTwo {
    static void methodOfClassTwo() {
        System.out.println("From Class Two");
    }
}
```

- **File Name:** Can be anything (`ClassOne.java`, `ClassTwo.java`, `Anything.java`)
- **Compilation:** `javac ClassOne.java` (or any chosen name)
- **.class Files Generated:** `ClassOne.class`, `ClassTwo.class`
- **Execution:** `java ClassOne` (since it has `main()`)

🔹 Rule #2: If One Class is `public`, File Name Must Match

```
public class ClassOne {
    public static void main(String[] args) {
        ClassTwo.methodOfClassTwo();
    }
}

class ClassTwo {
    static void methodOfClassTwo() {
        System.out.println("From Class Two");
    }
}
```

- ✅ **File Name:** `ClassOne.java` only
- ❌ Any other file name → Compile-time error
- **Compile:** `javac ClassOne.java`
- **Run:** `java ClassOne`

🔹 Rule #3: If a Different Class is `public`, File Name Must Match That

```
class ClassOne {
    public static void main(String[] args) {
        ClassTwo.methodOfClassTwo();
    }
}

public class ClassTwo {
    static void methodOfClassTwo() {
        System.out.println("From Class Two");
    }
}
```

- ✅ **File Name:** `ClassTwo.java`
- **Compile:** `javac ClassTwo.java`
- **Run:** `java ClassOne` (only `ClassOne` has a `main()` method)
- ❌ `java ClassTwo` → Error: `Main method not found`

🔹 Rule #4: Two Public Classes in One File → Not Allowed

```
public class ClassOne {
    public static void main(String[] args) {
        ClassTwo.methodOfClassTwo();
    }
}

public class ClassTwo {
    static void methodOfClassTwo() {
        System.out.println("From Class Two");
    }
}
```

- ❌ Compile-time error: *"The public type ClassTwo must be defined in its own file"*
- ✅ Only **one public class per `.java` file** is allowed

🔹 Rule #5: Multiple Classes with Their Own `main()` Methods

```
class ClassOne {
    public static void main(String[] args) {
        System.out.println("From Class One");
    }
}
```

```
class ClassTwo {
    public static void main(String[] args) {
        System.out.println("From Class Two");
    }
}
```

- ✓ **File Name:** Anything (`ClassOne.java`, `ClassTwo.java`, etc.)
- ✓ **Compile:** `javac ClassOne.java`
- ✓ **Run:**
  - `java ClassOne` → `From Class One`
  - `java ClassTwo` → `From Class Two`

☐ Summary Table

Condition	File Name	Compile	Run	Notes
No public class	Any	Yes	Class with `main()`	Multiple classes OK
One public class	Must match class name	Yes	Class with `main()`	Only one public allowed
Two public classes	✗ Not allowed	✗ Error	✗	Breaks rule
One `main()` method	Run that class	Yes	Yes	Entry point required
Multiple `main()` methods	Run any class	Yes	Yes	Independent executables

---

✓ Tips

- `.java` file can contain multiple classes but **only one `public` class**, and file name must match it.
- You can compile **any `.java` file**, but to run it, the class **must have `public static void main(String[] args)`.**
- Each class in the file is compiled into a separate `.class` file.
- Use multiple `main()` methods to **test individual components independently**.

Awesome comprehensive breakdown of the `final` keyword in Java! Here's a clear, organized summary and explanation of those **10 important points about `final`** that every Java programmer should know — with code highlights and key takeaways:

Java `final` Keyword: What You Must Know

What does `final` mean?

- It **prevents further modification**.
- Used with **classes, methods, and variables**.
- Implies the entity is **complete, locked, and cannot be changed**.

---

1) `final class`

- **Cannot be subclassed or extended.**

```
final class FinalClass { }
class SubClass extends FinalClass { }  // Compile error!
```

---

2) `final method`

- **Cannot be overridden by subclasses.**
- BUT, **final methods can be overloaded** in the same class, and overloaded methods **can be overridden**.

```
class SuperClass {
    final void methodOne() { }
    void methodOne(int i) { } // overload
}

class SubClass extends SuperClass {
    @Override
    void methodOne(int i) { }  // override allowed
    // void methodOne() { }    // override NOT allowed (final)
}
```

---

3) `final variable`

- Once initialized, **value cannot be changed** (immutable).
- But can be used to initialize other variables.

```
final int i = 10;
i = 20;  // Compile error
int j = i; // allowed
```

---

4) `final` with arrays

- The **reference** to the array is final (cannot reassign),
- but **array elements can be modified**.

```
final int[] arr = new int[10];
arr[0] = 100;  // allowed
arr = new int[20];  // Compile error
```

---

5) `final` reference variables

- You **cannot reassign** the reference to a new object,
- but you **can modify the object's internal state**.

```
final A a = new A();
a.i = 50;   // allowed
a = new A(); // Compile error
```

---

6) `final` variables can be static, non-static, or local

- Once initialized, **cannot be reassigned anywhere**.
- Applies to static variables, instance variables, and method parameters.

```
static final int i = 10;
final int j = 20;
void method(final int k) {
    k = 30;  // Compile error
}
```

---

7) Blank final variables

- **Final instance variables must be initialized explicitly**,
- either at declaration, in constructor(s), or instance initialization blocks.
- No default initialization for final fields!

```
class A {
    final int j; // blank final

    A() {
        j = 20;  // must be initialized here
    }
}
```

---

8) Initialization of final instance variables

- Must be initialized at declaration OR

- in **every constructor OR**
- in an instance initialization block (IIB).

```
class A {
    final int i;

    { i = 30; }  // IIB

    A() { /* i must be initialized if not done before */ }
    A(int x) { i = x; }
}
```

---

9) Initialization of final static variables

- Must be initialized at declaration OR
- in a static initialization block (SIB).
- **Cannot be initialized in constructors.**

```
class A {
    static final int i;

    static {
        i = 30;
    }
}
```

---

10) `final static` variables are constants

- **Static final variables are constants**—shared among all objects, cannot be changed once set.
- Interface fields are implicitly `public static final`.

```
static final int MAX_VALUE = 100;
// This is a constant shared by all instances
```

Bonus: `abstract` vs `final`

- A class or method **cannot be both `abstract` and `final`** — contradictory concepts:
  - `abstract` means "must be overridden,"
  - `final` means "cannot be overridden."

```
final abstract class MyClass {}  // Compile error
final abstract void method();   // Compile error
```

Summary Table

Usage	Effect	Where to Initialize
`final class`	No subclassing	N/A
`final method`	No overriding	N/A
`final variable`	Immutable value	At declaration, constructor, or init block
`final static variable`	Constant value, shared by all objects	At declaration or static init block
Blank final variable	Must be initialized explicitly	Constructor or init block

---

Great question! Understanding how arrays are stored in memory is key to grasping Java's memory model and performance behavior.

How Are Arrays Stored in Memory in Java?

1. Arrays are objects in Java

- Every array in Java is an object, regardless of whether it's an array of primitives (e.g., `int[]`) or objects (e.g., `String[]`).
- This means every array has:
  - A header with metadata (like the array length and type information).
  - A contiguous block of memory storing the actual elements.

---

2. Memory layout of arrays

- **Reference to array object**: When you declare an array variable like `int[] arr;`, it stores a **reference** (pointer) to the actual array object in the heap.
- **Array object in the heap**: The actual array object resides in the **heap memory**, not on the stack.
- **Contiguous block for elements**: Inside the heap, the elements of the array are stored in a **contiguous block of memory**.
  - For primitives (like `int`, `double`), the actual values are stored one after another.
  - For object arrays, the contiguous block stores **references** to the objects, not the objects themselves.

---

3. Example

```
int[] arr = new int[5];
```

- `arr` is a reference variable stored on the stack (if local variable).
- The actual array object of 5 integers is created on the heap.
- The 5 `int` values are stored in a contiguous block of memory inside the heap.
- The `arr` reference points to the start of that memory block.

---

4. Visualizing memory

```
Stack                          Heap
+------------------+          +------------------------+
| arr (reference)  |   --->   | Array object header    |
+------------------+          | length = 5             |
                              | int[0] int[1] int[2] int[3] int[4] |
                              +------------------------+
```

---

5. Object arrays example

```
String[] strArr = new String[3];
strArr[0] = "Hello";
strArr[1] = "World";
```

- `strArr` references the array object in the heap.
- The array object contains **3 references**, each pointing to `String` objects elsewhere in the heap.

```
Stack                          Heap
+------------------+          +------------------------+
| strArr           |   --->   | Array object header    |
+------------------+          | length = 3             |
                              | ref to "Hello"         |
                              | ref to "World"         |
                              | null                   |
                              +------------------------+
                              +------------------+
                              | "Hello" String obj|
                              +------------------+
                              +------------------+
                              | "World" String obj|
                              +------------------+
```

---

6. Summary

Aspect	Details
Array variable	Reference stored on stack (if local variable)
Array object	Stored in heap
Array elements	Stored in contiguous memory block
Primitive arrays	Actual primitive values stored contiguously
Object arrays	Contiguous references to objects

Sure! Passing arrays as parameters and returning arrays from methods is a very common pattern in Java. Here's a clear explanation with examples:

Arrays as Parameters and Return Types of Methods in Java

---

1. Passing Arrays as Parameters

You can pass an array to a method just like any other variable. The method receives a **reference** to the original array, so changes made inside the method affect the original array.

Example:

```
public class ArrayExample {

    // Method to print all elements of the array
    static void printArray(int[] arr) {
        for (int element : arr) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    // Method to modify the array elements by doubling them
    static void doubleArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            arr[i] = arr[i] * 2;
        }
    }

    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        System.out.println("Original array:");
        printArray(numbers);

        doubleArray(numbers);

        System.out.println("Modified array after doubling:");
        printArray(numbers);
    }
}
```

Output:

```
Original array:
1 2 3 4 5
Modified array after doubling:
2 4 6 8 10
```

Explanation:

- `numbers` array is passed to both `printArray` and `doubleArray` methods.
- `doubleArray` modifies the original array because the array reference is passed, not a copy.

2. Returning Arrays from Methods

You can create and return an array from a method.

Example:

```
public class ArrayReturnExample {

    // Method to create and return an array of size n filled with squares of indices
    static int[] createSquaresArray(int n) {
        int[] squares = new int[n];
        for (int i = 0; i < n; i++) {
            squares[i] = i * i;
        }
        return squares;
    }

    public static void main(String[] args) {
        int[] squaresArray = createSquaresArray(5);

        System.out.println("Squares array:");
        for (int val : squaresArray) {
            System.out.print(val + " ");
        }
    }
}
```

Output:

```
Squares array:
0 1 4 9 16
```

Explanation:

- The method `createSquaresArray` creates a new array and returns it.
- The caller receives the array reference and can use it.

3. Important Notes

- Since arrays are objects, passing an array to a method passes a reference to the original array.
- Any modifications inside the method affect the original array.
- If you want to avoid that, you have to clone the array before passing it (`arr.clone()`).
- Methods can return arrays, and the returned array is usually created inside the method.

Great list! These 10 observations cover some essential and tricky aspects of arrays in Java that often pop up in interviews or certification exams. Let me summarize and clarify them neatly for you:

10 Interesting Observations About Arrays in Java

1) Negative Size Throws Runtime Exception

- Array size cannot be negative.
- No compile-time error if size is negative.
- **Runtime:** `NegativeArraySizeException`.

```
int[] arr = new int[-5];  // Compiles fine, but throws NegativeArraySizeException at
runtime.
```

2) Array Size Must Be an Integer or Expression Resulting in Integer

- Size must be an integer or expression evaluated to int.
- Auto-widening allowed (`byte` to `int`).
- Floating types like `double` not allowed.

```
int[] arr1 = new int[10 + 5];       // Valid
int[] arr2 = new int[(byte)10];     // Valid (auto-widening)
```

```
int[] arr3 = new int[10.25];        // Compile-time error
```

---

3) Declaration and Instantiation Types Must Match Exactly (Except Auto-upcasting)

- No auto-boxing/unboxing or widening conversions allowed for array types.
- Only auto-upcasting allowed (e.g., `String[]` to `Object[]`).

```
Integer[] I = new int[5];          // Compile-time error (no auto-boxing)
int[] i = new Integer[10];         // Compile-time error (no auto-unboxing)
long[] l = new byte[10];           // Compile-time error (no auto-widening)
Object[] o = new String[10];       // Valid (auto-upcasting)
```

---

4) Array Element Type Compatibility Checked at Runtime

- You can assign a subtype array to a supertype array reference.
- But inserting incompatible elements throws `ArrayStoreException` at runtime.

```
Object[] o = new String[10];
o[2] = "java";    // OK
o[5] = 20;        // Runtime: ArrayStoreException
```

---

5) Array Initialization Without `new` Must Be at Declaration

- You can only initialize an array with `{ ... }` syntax at declaration.
- To assign later, you must use `new` keyword.

```
int[] i = {1, 2, 3, 4};         // Valid
i = {1, 2, 3, 4};               // Compile-time error
i = new int[]{1, 2, 3, 4};      // Valid
```

---

6) Multiple Ways to Declare Multi-Dimensional Arrays

```
int[][] twoD1;      // Standard declaration
int[] twoD2[];      // Also valid
int[][][] threeD1;  // Standard 3D array
int[] threeD2[][];  // Also valid
```

---

7) You Cannot Skip Dimensions When Creating Multi-Dimensional Arrays

- When creating arrays, you can't leave an empty dimension in the middle.

```
int[][][] arr = new int[10][][10];  // Compile-time error
int[][][] arr2 = new int[][10][];   // Compile-time error
int[][][] arr3 = new int[][][10];   // Compile-time error
```

---

8) Anonymous Arrays Are Allowed

- Arrays can be created without references, useful for method arguments or quick use.

```
System.out.println(new int[]{1, 2, 3}.length);     // Outputs: 3
System.out.println(new int[]{47, 21, 58, 98}[1]);  // Outputs: 21
```

---

9) Assignment Between Arrays Only Checks Type, Not Size

```
int[] a = new int[10];
int[] b = new int[100];
double[] c = new double[20];

a = b;    // Valid, same type
b = c;    // Compile-time error, incompatible types
```

---

10) Array Size is Fixed

- Once created, array size cannot be changed.
- You can modify elements, but not add or remove elements.
- This is a key limitation of arrays compared to collections like ArrayList.

Absolutely! Here's a comprehensive list of **key points about Strings in Java** that every programmer should know — perfect for interviews or certifications.

Things You Should Know About Strings in Java

1) String is Immutable

- Once a `String` object is created, it cannot be changed.
- Any operation that seems to modify a string actually creates a new string object.

```
String s = "Hello";
s = s + " World";   // Creates a new String object, original "Hello" remains unchanged
```

---

2) String Pool (String Interning)

- String literals are stored in a special memory area called the **String Pool**.
- When you create a string literal, JVM checks if it already exists in the pool.
- If yes, it returns the reference, else it creates a new one.
- This saves memory and improves performance.

```
String s1 = "Java";
String s2 = "Java";
System.out.println(s1 == s2);  // true, both refer to same object in the pool
```

---

3) `new String()` Always Creates a New Object

- Using `new String("text")` creates a new object in heap, not in String pool.
- Even if the same literal exists in the pool.

```
String s1 = "Java";
String s2 = new String("Java");
System.out.println(s1 == s2);  // false, different objects
```

---

4) Strings are `final` Class

- You cannot extend the `String` class.
- This helps in preserving immutability and security.

---

5) String Comparison: `==` vs `.equals()`

- `==` compares **references** (memory addresses).
- `.equals()` compares **contents** (characters).
- Always use `.equals()` to compare string contents.

```
String s1 = new String("abc");
String s2 = new String("abc");
System.out.println(s1 == s2);         // false
System.out.println(s1.equals(s2));    // true
```

---

6) `intern()` Method

- It returns a canonical representation for the string object.
- If the pool already contains a string equal to this one, it returns the reference from the pool.
- Otherwise, adds the string to the pool and returns its reference.

```
String s1 = new String("Hello");
String s2 = s1.intern();
String s3 = "Hello";
System.out.println(s2 == s3);  // true
```

---

7) String Concatenation and Performance

- Using `+` operator creates many temporary String objects (because strings are immutable).
- For heavy concatenation inside loops, use **`StringBuilder`** or **`StringBuffer`**.

```
StringBuilder sb = new StringBuilder();
for(int i=0; i<1000; i++) {
    sb.append(i);
}
String result = sb.toString();
```

---

8) Strings can be Used as Keys in Hash-Based Collections

- `String` class overrides `hashCode()` and `equals()` correctly.
- Makes it a good choice as keys in HashMap, HashSet.

---

9) String Methods Are Powerful

- `substring()`, `charAt()`, `indexOf()`, `split()`, `replace()`, `toLowerCase()`, `toUpperCase()`, `trim()` — all very useful.
- Remember `substring()` returns a new string; it does not modify the original string.

---

10) Strings Are Serializable and Comparable

- Implements `Serializable` interface for easy storage/transfer.
- Implements `Comparable<String>` for natural lexicographic ordering.

Sure! Here's a clear explanation of **throwing and re-throwing exceptions in Java**, with examples to help you understand the concepts:

Throwing an Exception in Java

- You **throw** an exception when you explicitly create and pass an exception object using the `throw` keyword.
- This is often done inside a method to indicate something went wrong.

Syntax:

```
throw new ExceptionType("Error message");
```

Example:

```
public class ThrowExample {
    public static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older");
        }
        System.out.println("Age is valid");
    }

    public static void main(String[] args) {
        checkAge(15); // This will throw IllegalArgumentException
    }
}
```

Re-Throwing an Exception in Java

- **Re-throwing** means catching an exception and then throwing it again.
- This is useful when you want to handle the exception partially (like logging) but still want the caller to handle it.

Why re-throw?

- To log or add context to the exception
- To perform some cleanup
- To transform the exception type

Example of re-throwing:

```
public class ReThrowExample {
    public static void riskyMethod() throws Exception {
        throw new Exception("Original Exception");
    }
    public static void wrapperMethod() throws Exception {
        try {
            riskyMethod();
        } catch (Exception e) {
            System.out.println("Logging exception: " + e.getMessage());
            throw e;  // Re-throwing the caught exception
        }
    }

    public static void main(String[] args) {
        try {
            wrapperMethod();
        } catch (Exception e) {
            System.out.println("Exception handled in main: " + e.getMessage());
        }
    }
}
```

Important Points:

- When you re-throw an exception, the original stack trace is preserved if you just use `throw e;`.
- If you throw a new exception (`throw new Exception(...)`), you lose the original stack trace unless you pass the caught exception as the cause.
- `throw new Exception("New message", e);  // Preserves cause`

Summary:

Concept	Description	Keyword Used
Throwing	Creating and throwing an exception explicitly.	`throw`
Re-Throwing	Catching and throwing the same exception again.	`throw e;`

Great topic! Here's a detailed explanation of **method overriding with the `throws` clause in Java**, including the rules, examples, and some important points:

Method Overriding and `throws` Clause in Java

When you **override** a method in a subclass, the overriding method can declare exceptions in its `throws` clause, but there are strict rules about what exceptions it can declare compared to the superclass method.

Rules for `throws` clause in method overriding:

1. **The overriding method cannot throw broader checked exceptions than the overridden method.**
   - If the superclass method declares a checked exception, the subclass overriding method can:
     - Throw the **same exception**, or
     - Throw **subclasses** of that exception, or
     - **Throw no checked exceptions** at all.
2. **The overriding method can throw any unchecked exceptions (RuntimeException or its subclasses) regardless of the superclass method.**
3. **If the superclass method does not declare any checked exceptions, the overriding method cannot declare any checked exceptions.**

Why these rules?

- These rules are there to maintain **polymorphism** and ensure the subclass method can be used anywhere the superclass method is expected without unexpected exceptions being thrown.

Example 1: Overriding with the same checked exception

```
class Parent {
    void show() throws IOException {
        // code
    }
}

class Child extends Parent {
    @Override
    void show() throws IOException {
        // overriding with the same exception
    }
}
```

Example 2: Overriding with subclass of the declared exception

```
class Parent {
    void show() throws Exception {
        // code
    }
}

class Child extends Parent {
    @Override
    void show() throws IOException {  // IOException is subclass of Exception
        // valid override
    }
}
```

Example 3: Overriding method cannot throw broader exception

```
class Parent {
    void show() throws IOException {
        // code
    }
}

class Child extends Parent {
    @Override
    void show() throws Exception {  // Compile error! Exception is broader than
IOException
        // invalid override
    }
}
```

Example 4: Overriding method throws no checked exception

```
class Parent {
    void show() throws IOException {
        // code
    }
}

class Child extends Parent {
    @Override
    void show() {
        // no exception thrown here — allowed
    }
}
```

Example 5: Overriding method throwing unchecked exceptions (allowed anytime)

```
class Parent {
    void show() {
        // no exceptions declared
    }
}

class Child extends Parent {
    @Override
    void show() throws ArithmeticException {  // RuntimeException subclass
        // allowed
    }
}
```

Summary Table

Superclass Method `throws`	Subclass Method `throws` Allowed?
None	No checked exceptions; any unchecked exceptions allowed
Checked Exception (E)	E or subclass of E, or none
Checked Exception (E)	Broader exception than E — **Compile-time error**
Unchecked Exception	Any checked or unchecked exceptions allowed (no restriction)

Absolutely! Here's a detailed explanation of each new Java String method from Java 8 to Java 17, along with separate, uniquely named Java programs demonstrating each method clearly.

Java 8 New String Method

1. `join()`

- **Purpose:** Joins multiple strings with a specified delimiter.
- **Overloads:** One accepts a delimiter and array of strings, another accepts a delimiter and an `Iterable` (like a List).

```
import java.util.Arrays;
import java.util.List;

public class Java8StringJoinDemo {
    public static void main(String[] args) {
        // Using join with array of strings
        String languages = String.join("_", "Java", "HTML", "Python", "CSS", "PHP");
        System.out.println(languages);  // Output: Java_HTML_Python_CSS_PHP

        // Using join with List
        List<String> languageList = Arrays.asList("Java", "HTML", "Python", "CSS",
"PHP");
        languages = String.join(", ", languageList);
        System.out.println(languages);  // Output: Java, HTML, Python, CSS, PHP
    }
}
```

Java 9 New String Methods

2. `chars()`

- **Purpose:** Returns an `IntStream` of the characters (UTF-16 code units) of the string.

```
public class Java9StringCharsDemo {
    public static void main(String[] args) {
        "String".chars().forEach(System.out::println);
        // Output: ASCII values of S, t, r, i, n, g
    }
}
```

3. `codePoints()`

- **Purpose:** Returns an `IntStream` of Unicode code points from the string (handles surrogate pairs properly).

```
public class Java9StringCodePointsDemo {
    public static void main(String[] args) {
        "String".codePoints().forEach(System.out::println);
        // Output: Same as chars() for basic ASCII, but handles Unicode correctly for
complex characters.
    }
}
```

Java 10 New String Methods

- **No new String methods introduced.**

Java 11 New String Methods

4. `isBlank()`

- **Purpose:** Checks if a string is empty or contains only whitespace characters.

```
public class Java11StringIsBlankDemo {
    public static void main(String[] args) {
        System.out.println("".isBlank());          // true
        System.out.println("   ".isBlank());       // true
        System.out.println("\t \n".isBlank());     // true
        System.out.println("Java".isBlank());      // false
    }
}
```

5. `lines()`

- **Purpose:** Returns a stream of lines from the string, splitting on line terminators.

```
public class Java11StringLinesDemo {
    public static void main(String[] args) {
        String multiline = "Line1\nLine2\nLine3";
        multiline.lines().forEach(System.out::println);
        // Prints each line separately
    }
}
```

6. `repeat(int count)`

- **Purpose:** Returns a new string consisting of the original string repeated `count` times.

```
public class Java11StringRepeatDemo {
    public static void main(String[] args) {
        System.out.println("abc".repeat(3));  // abcabcabc
        System.out.println("1".repeat(5));    // 11111
    }
}
```

7. `strip()`

- **Purpose:** Removes leading and trailing whitespaces (Unicode aware).

```
public class Java11StringStripDemo {
    public static void main(String[] args) {
        System.out.println("   hello  ".strip()); // "hello"
    }
}
```

8. `stripLeading()`

- **Purpose:** Removes only leading whitespaces.

```
public class Java11StringStripLeadingDemo {
    public static void main(String[] args) {
        System.out.println("   hello  ".stripLeading()); // "hello  "
    }
}
```

9. `stripTrailing()`

- **Purpose:** Removes only trailing whitespaces.

```
public class Java11StringStripTrailingDemo {
    public static void main(String[] args) {
        System.out.println("   hello  ".stripTrailing()); // "   hello"
    }
}
```

Java 12 New String Methods

10. `indent(int n)`

- **Purpose:** Adds indentation (or removes if negative) for each line of the string.

```
public class Java12StringIndentDemo {
    public static void main(String[] args) {
        System.out.println("Hello\nWorld".indent(4));
        /*
         Output:
             Hello
             World
         */
    }
}
```

11. `transform(Function<String, R> f)`

- **Purpose:** Applies a function to the string and returns the result.

```
public class Java12StringTransformDemo {
    public static void main(String[] args) {
        String result = "java".transform(String::toUpperCase);
        System.out.println(result);  // JAVA

        String combined = "abc".transform(s -> s + "xyz").transform(String::toUpperCase);
        System.out.println(combined); // ABCXYZ
    }
}
```

12. `describeConstable()`

- **Purpose:** Returns an Optional describing the string as a constant (related to Java's constant API).

```
public class Java12StringDescribeConstableDemo {
    public static void main(String[] args) {
        System.out.println("Hello".describeConstable().get()); // Hello
    }
}
```

13. `resolveConstantDesc(MethodHandles.Lookup lookup)`

- **Purpose:** Resolves the string constant descriptor.

```
import java.lang.invoke.MethodHandles;

public class Java12StringResolveConstantDescDemo {
    public static void main(String[] args) throws Throwable {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        System.out.println("Hello".resolveConstantDesc(lookup)); // Hello
    }
}
```

Java 13 New String Methods

- **No new String methods introduced.**

Java 14 New String Methods

- **No new String methods introduced.**

Java 15 New String Methods

14. `formatted(Object... args)`

- **Purpose:** Formats the string with the supplied arguments, similar to `String.format`.

```
public class Java15StringFormattedDemo {
    public static void main(String[] args) {
        System.out.println("Hello, %s! You have %d new messages.".formatted("Alice", 5));
        // Output: Hello, Alice! You have 5 new messages.
    }
}
```

15. `stripIndent()`

- **Purpose:** Removes common leading whitespace from every line.

```
public class Java15StringStripIndentDemo {
    public static void main(String[] args) {
        String indented = "    line1\n    line2\n    line3";
        System.out.println(indented.stripIndent());
        /*
        Output:
        line1
        line2
        line3
        */
    }
}
```

16. `translateEscapes()`

- **Purpose:** Converts escape sequences in the string to their corresponding characters.

```
public class Java15StringTranslateEscapesDemo {
    public static void main(String[] args) {
        String str = "Tab \\t New Line \\n Single Quote \\' Double Quote \\\"";
        System.out.println(str.translateEscapes());
        /*
        Output:
        Tab    New Line
         Single Quote ' Double Quote "
        */
    }
}
```

Java 16 New String Methods

- **No new String methods introduced.**

Java 17 New String Methods

- **No new String methods introduced.**
