**What are Wrapper Classes?**

* In Java, **primitive data types** (int, char, double, etc.) are **not objects**.
* But **everything in Java works with objects** (for example: Collections like ArrayList, HashMap cannot store primitives directly).
* To solve this → Java provides **Wrapper Classes** (object representation of primitive data types).

👉 Primitive → Wrapper (Object)

| **Primitive** | **Wrapper Class** |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

**2. 🔹 Why do we need Wrapper Classes?**

✅ **Object-Oriented features** – Java works mainly with objects.  
✅ **Collections Framework** – ArrayList<Integer> instead of ArrayList<int> (because generics need objects).  
✅ **Utility Methods** – Wrapper classes contain useful methods (parseInt(), valueOf(), compare(), etc.).  
✅ **Type conversion** – Convert between primitives and Strings easily.  
✅ **Null values** – Unlike primitives, wrapper objects can hold null.

**3. 🔹 Example of Wrapper Classes**

public class WrapperExample {

public static void main(String[] args) {

int a = 10; // primitive

Integer obj = Integer.valueOf(a); // manual boxing

System.out.println("Wrapper object: " + obj);

double d = 5.67;

Double dObj = Double.valueOf(d); // boxing

System.out.println("Double object: " + dObj);

char c = 'A';

Character cObj = Character.valueOf(c); // boxing

System.out.println("Character object: " + cObj);

}

}

**4. 🔹 Autoboxing (Primitive → Wrapper automatically)**

👉 Since **Java 5**, you don’t need to manually use valueOf().  
The compiler **automatically converts** a primitive into its wrapper object.

**Example:**

public class AutoBoxingDemo {

public static void main(String[] args) {

int a = 100;

// Autoboxing: int → Integer automatically

Integer obj = a;

System.out.println("Primitive: " + a);

System.out.println("Wrapper (Autoboxed): " + obj);

// Works directly inside Collections

ArrayList<Integer> list = new ArrayList<>();

list.add(25); // int will be autoboxed to Integer

list.add(30);

System.out.println("ArrayList: " + list);

}

}

✅ Compiler does → Integer.valueOf(a) behind the scenes.

**5. 🔹 Unboxing (Wrapper → Primitive automatically)**

👉 Opposite of autoboxing.  
The compiler **automatically converts** a wrapper object back to primitive.

**Example:**

public class UnboxingDemo {

public static void main(String[] args) {

Integer obj = 200; // Autoboxing

// Unboxing: Integer → int automatically

int a = obj;

System.out.println("Wrapper: " + obj);

System.out.println("Primitive (Unboxed): " + a);

}

}

✅ Compiler does → obj.intValue() behind the scenes.

**6. 🔹 Autoboxing + Unboxing Together**

public class AutoUnboxDemo {

public static void main(String[] args) {

ArrayList<Double> marks = new ArrayList<>();

// Autoboxing

marks.add(98.5); // double → Double

marks.add(87.0);

// Unboxing while fetching

double m1 = marks.get(0); // Double → double

double m2 = marks.get(1);

System.out.println("Marks: " + m1 + ", " + m2);

}

}

**7. 🔹 Utility Methods in Wrapper Classes**

Each wrapper class provides **helper methods**. Examples:

public class WrapperMethods {

public static void main(String[] args) {

// Converting String → int

int num = Integer.parseInt("123");

System.out.println("String to int: " + num);

// Converting int → String

String s = Integer.toString(456);

System.out.println("int to String: " + s);

// Comparing values

Integer a = 10, b = 20;

System.out.println("Compare: " + a.compareTo(b)); // -1

// Max and Min values

System.out.println("Integer Max: " + Integer.MAX\_VALUE);

System.out.println("Integer Min: " + Integer.MIN\_VALUE);

// Character utility

System.out.println("Is Digit? " + Character.isDigit('5')); // true

System.out.println("Is Letter? " + Character.isLetter('A')); // true

}

}

**8. 🔹 Key Points to Remember**

* Autoboxing/unboxing happens automatically since **Java 5**.
* Wrapper objects are **immutable** (once created, cannot change value).
* Use **== carefully** → compares references for Wrapper objects, not values (except small integers -128 to 127 due to caching).
* For value comparison, always use .equals().

Example:

Integer a = 100;

Integer b = 100;

System.out.println(a == b); // true (cached values)

Integer x = 200;

Integer y = 200;

System.out.println(x == y); // false (different objects)

System.out.println(x.equals(y)); // true (same value)

✅ **Summary:**

* Wrapper classes = object representation of primitives.
* Needed for Collections, OOP features, utility methods.
* Autoboxing = primitive → wrapper (automatic).
* Unboxing = wrapper → primitive (automatic).
* Always use .equals() for value comparison.

## ****What is Exception Handling?****

* An **exception** is an **unexpected event** that disrupts the normal flow of a program.
* **Exception Handling** in Java is a **mechanism to handle runtime errors** so that the program can continue execution instead of crashing.

## 🔹 ****Types of Errors in Java****

1. **Compile-time Errors**
   * Syntax mistakes (like missing ;, wrong keywords).
   * Detected by the compiler.
2. **Runtime Errors (Exceptions)**
   * Occur during execution (like division by zero, array index out of bounds).
   * These are handled using **exception handling**.
3. **Logical Errors**
   * Program runs but gives wrong output due to wrong logic.
   * Cannot be handled by compiler or exception mechanism.

## 🔹 ****Exception Hierarchy****

In Java, all exceptions are objects of classes derived from **Throwable**:

Throwable

├── Error (Serious issues, cannot be handled by program: OutOfMemoryError)

└── Exception

├── Checked Exceptions (Compile-time checked: IOException, SQLException)

└── Unchecked Exceptions (Runtime: ArithmeticException, NullPointerException)

## 🔹 ****Why Exception Handling?****

* To maintain **normal flow** of program.
* To provide **meaningful error messages**.
* To **separate error-handling code** from normal business logic.

## 🔹 ****5 Keywords used in Exception Handling****

1. **try** → Block of code to be monitored for exceptions.
2. **catch** → Block of code that handles exception.
3. **finally** → Block that always executes (cleanup code like closing file, DB connection).
4. **throw** → Used to explicitly throw an exception.
5. **throws** → Used in method signature to declare exceptions.

# 🔹 **Basic Example**

public class ExceptionExample {

public static void main(String[] args) {

try {

int a = 10 / 0; // risky code

} catch (ArithmeticException e) {

System.out.println("Error: Cannot divide by zero!");

} finally {

System.out.println("Finally block always executes.");

}

System.out.println("Program continues...");

}

}

**Output:**

Error: Cannot divide by zero!

Finally block always executes.

Program continues...

## 🔹 ****Checked vs Unchecked Exception****

* **Checked**: Must be handled during compile-time.  
  Example: IOException, SQLException.
* **Unchecked**: Occur at runtime, no compiler checking.  
  Example: NullPointerException, ArithmeticException.

## 🔹 ****throw vs throws****

// throw -> to throw exception explicitly

throw new IOException("File not found");

// throws -> declare exceptions in method signature

void readFile() throws IOException { }

## 🔹 ****Multiple Catch Blocks****

try {

int arr[] = new int[5];

arr[10] = 50; // ArrayIndexOutOfBounds

} catch (ArithmeticException e) {

System.out.println("Arithmetic Exception");

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array Index Error");

} catch (Exception e) {

System.out.println("General Exception");

}

## 🔹 ****Custom Exceptions (User-defined)****

You can define your own exceptions by extending Exception:

class MyException extends Exception {

public MyException(String msg) {

super(msg);

}

}

public class Test {

public static void main(String[] args) {

try {

throw new MyException("This is a custom exception!");

} catch (MyException e) {

System.out.println("Caught: " + e.getMessage());

}

}

}

✅ **In short:** Exception handling in Java = try → catch → finally → throw → throws.  
It ensures smooth execution, prevents program crash, and makes debugging easier.

## try

**What it is:**  
Marks a block of “risky” code that might throw an exception.

**Forms:**

1. try { ... } catch (...) { ... }
2. try { ... } finally { ... } (no catch)
3. try (resources) { ... } catch (...) { ... } finally { ... } (try-with-resources)

**Key rules & edges:**

* A try must be followed by **at least one** catch **or** a finally.
* If code in try throws and no matching catch exists, the exception **propagates** after executing finally.
* return, break, continue, and even a thrown exception inside try **still run** finally (unless the JVM halts, e.g., System.exit(0) or fatal error).
* try-with-resources auto-closes resources that implement **AutoCloseable** in **reverse** declaration order, and captures “close” failures as **suppressed exceptions** (Java 7+).

**Example (plain try–catch–finally):**

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Can't divide by zero");

} finally {

System.out.println("cleanup happens here");

}

**Example (try-with-resources):**

try (java.io.BufferedReader br = new java.io.BufferedReader(

new java.io.FileReader("data.txt"))) {

System.out.println(br.readLine());

} // br.close() is auto-called here

**Best practices:**

* Keep the try block **small**—only wrap the statements that can actually throw.
* Prefer **try-with-resources** for anything you must close (files, streams, sockets, JDBC).

## catch

**What it is:**  
Handles a thrown exception type.

**Key rules:**

* **Order matters.** Catch **more specific** exceptions **before** more general ones; otherwise the general one makes the specific unreachable (compile error).
* You can chain multiple catches, or use **multi-catch** with | (Java 7+).
* The caught exception variable is **effectively final**; you can’t reassign it.
* If none of the catch clauses match, the exception continues upward.

**Examples:**

try {

String s = null;

s.length(); // NullPointerException

} catch (NullPointerException e) { // specific first

System.out.println("Null!");

} catch (RuntimeException e) { // then general

System.out.println("Runtime problem");

}

**Multi-catch (OR types):**

try {

int[] a = new int[2];

a[5] = 99; // ArrayIndexOutOfBoundsException

} catch (IllegalArgumentException | ArrayIndexOutOfBoundsException e) {

System.out.println("Bad argument or bad index: " + e.getMessage());

// e = new Exception(); // ❌ not allowed (effectively final)

}

**Re-throwing and type inference (Java 7+):**

void f() throws java.io.IOException {

try {

risky(); // may throw IOException or SQLException

} catch (Exception e) {

// With analysis, compiler may know e is IOException|SQLException.

// If you just `throw e;`, your method’s throws must cover possible types.

throw e;

}

}

**Best practices:**

* Catch only what you can **meaningfully handle**. Otherwise, let it propagate.
* Log with context and **preserve the cause** if you wrap (new X("msg", e)).

## finally

**What it is:**  
A block that **always** runs after try (and after catch, if any), typically for cleanup.

**Runs when?**

* On normal completion of try.
* When a catch handles an exception.
* Even if try or catch has a return (**finally still runs**).
* **Does not** run if the JVM **terminates** (e.g., System.exit(0)), power loss, or catastrophic errors.

**Dangerous edge cases:**

* **Returning** from finally is a **bad idea**—it **overrides** earlier returns and even **swallows** exceptions.
* Throwing an exception in finally can **mask** the original failure.

**Examples:**

static int g() {

try {

return 1;

} finally {

System.out.println("finally runs");

}

}

static int h() {

try {

return 1;

} finally {

return 2; // ❌ avoid: overrides 1, hard to debug

}

}

**Best practices:**

* Prefer **try-with-resources** for closing resources.
* In finally, do minimal, failure-safe cleanup. Don’t throw.

## throw

**What it is:**  
Creates and **immediately throws** a specific exception **instance**.

**Syntax:**

throw new IllegalArgumentException("age must be >= 0");

**Where used:**

* Inside methods/blocks to signal a problem right away (precondition checks).
* Inside catch blocks to **wrap** and rethrow (exception chaining).

**Checked vs unchecked with throw:**

* You can throw either kind.
* If you throw a **checked** exception, the method must **declare** it with throws (or handle it in a try/catch).

**Chaining the cause:**

try {

parseConfig();

} catch (IOException e) {

throw new RuntimeException("Failed to load config", e); // preserves cause

}

**Best practices:**

* Prefer **specific** exception types (IllegalArgumentException, IllegalStateException, NoSuchElementException, etc.).
* Include a **clear message**; add the **cause** when rethrowing.

## throws

**What it is:**  
A **method (or constructor) declaration** that **advertises** it may throw certain exceptions.

**Syntax:**

void load() throws IOException, ParseException {

// ...

}

**Rules:**

* You **must** declare **checked** exceptions you don’t handle.
* You **may** declare unchecked ones, but it’s optional.
* **Overriding methods**:
  + Cannot declare **broader** checked exceptions than the overridden method.
  + May declare a **subset** (narrower) or **none**.
  + Unchecked exceptions aren’t restricted by throws.
* **Constructors** also use throws with the same rules.

**Examples:**

class A {

void work() throws IOException {}

}

class B extends A {

@Override

void work() /\* throws Exception \*/ { // ❌ broader than IOException

// OK to declare nothing or narrower:

// void work() throws FileNotFoundException { ... } // ✅ narrower

}

}

**When to use:**

* Library or service method that **can’t** handle the error meaningfully (let caller decide).
* APIs where callers need to **react** (e.g., IO failures, validation).

# Putting it together: propagation & flow

**Propagation (stack unwinding):**

* If a method throws and doesn’t catch, the exception **bubbles up** to the caller, and so on, until:
  + it’s caught, or
  + it reaches main → default handler prints stack trace → program ends.

**Stack trace & diagnostics:**

try {

faulty();

} catch (Exception e) {

e.printStackTrace(); // full trace

System.err.println(e.getMessage()); // short message

Throwable cause = e.getCause(); // original cause if chained

}

# Custom exceptions

**Checked**: extend Exception (forces callers to handle/declare).  
**Unchecked**: extend RuntimeException (for programming-error conditions).

class AgeTooLowException extends Exception { // checked

public AgeTooLowException(String msg) { super(msg); }

}

class ConfigMissingException extends RuntimeException { // unchecked

public ConfigMissingException(String msg, Throwable cause) { super(msg, cause); }

}

**Guidelines:**

* Name ends with **Exception**.
* Include **constructors** with message and (message, cause).
* Use **checked** when callers are expected to **recover**; use **unchecked** for **programming errors** (illegal state/arguments).

## try-with-resources

**Why:** auto-close + suppressed exceptions tracking.

**Suppressed exceptions:** if both the body and close() throw, the body’s exception wins; the close() exception is **suppressed** and attached.

try (MyRes r = new MyRes()) {

r.doWork(); // throws A

} catch (Exception e) {

for (Throwable t : e.getSuppressed()) {

System.err.println("Suppressed: " + t);

}

}

**Custom resource:**

class MyRes implements AutoCloseable {

public void close() throws Exception {

// release

}

}

# Common pitfalls & pro tips

* ❌ **Swallowing exceptions**:
* try { ... } catch (Exception e) { } // bad: hides problems

✅ At least log + context, or rethrow.

* ❌ **Returning from finally** (masks results/exceptions).
* ✅ Use **multi-catch** to avoid duplicate code; keep handlers cohesive.
* ✅ **Wrap** low-level exceptions with high-level context (and a cause).
* ✅ Validate inputs; for bad arguments, prefer **unchecked** (IllegalArgumentException).
* ✅ Keep exception messages **actionable**: what failed, what was expected, key identifiers.

# Mini cheat sheet

* **try** → mark risky code (optionally with resources).
* **catch** → handle specific problems; specific → general; can multi-catch.
* **finally** → always cleanup; avoid returns/throws here.
* **throw** → raise a specific exception now.
* **throws** → advertise that a method may throw (esp. checked).