Last Minute Revision of Java

✅ Chapter 1: Breaking the Surface

- **Java Flow:** Source code → `javac` compiler → bytecode → JVM executes it.
- **Bytecode:** Platform-independent.
- **JVM (Java Virtual Machine):** Executes bytecode; enables portability and performance (HotSpot optimization).
- **Java vs Other Languages:** Almost as fast as C/Rust, but uses more memory.
- **Compiler vs JVM:**
  - Compiler = First line of defense (type safety, access control).
  - JVM = Executes code, does runtime checks (e.g., dynamic binding).

✅ Chapter 2: A Trip to Objectville

- **OO Benefits:**
  - Encapsulation: Data + methods together.
  - Extensibility without touching tested code.
  - Reusability.
- **Object Design:**
  - Think: What the object **knows** (instance variables) and **does** (methods).
- **Class vs Object:**
  - Class = Blueprint.
  - Object = Instance of a class.

✅ Chapter 3: Know Your Variables

- **Primitive vs Object Reference:**
  - Primitive: Direct value.

- Reference: Points to an object (like a remote).

- **Dot Operator:** Used to access methods/fields → `myDog.bark();`
- **Type Casting & Spillage:** `int x = 24; byte b = x;` ❌ Compiler blocks this due to potential overflow.
- **Arrays:** Always objects, even if holding primitives.
- **Strings:** Special object, behaves like a primitive (e.g., `String name = "Avish";`).

✅ Chapter 4: How Objects Behave

- **Pass-by-Value:** Java always passes copies of variables.
  - For objects: copy of reference, not the actual object.
- **Encapsulation:** Make instance variables private, use getters/setters.
- **Instance vs Local Variables:**
  - Instance: Belongs to object; gets default value.
  - Local: Inside method; must be initialized.
- **Comparison:**
  - `==`: Compares references or primitive values.
  - `.equals()`: Compares object content.

✅ Chapter 5: Extra-Strength Methods

- **Test-Driven Development (TDD):**
  - Write test code → then implementation.
- **SimpleStartup Test Drive:** Use assertions to test methods like `checkYourself(int guess)`.
- **Class Structure Tips:**
  - Start with: prep code → test code → implementation.
  - Keep running tests after every code change.

✅ Chapter 6: Using the Java Library

- **ArrayList:**
  - Dynamically resizes.
  - Key methods: `add()`, `remove()`, `indexOf()`, `isEmpty()`, `size()`.
  - Holds only objects (primitives are autoboxed).
- **Packages:** Group related classes; use `import` if not in `java.lang`.
- **Short-Circuit Logic:**
  - `&&`, `||`: Stop early (efficient, safe).
  - `&`, `|`: Always evaluate both sides.

✅ Chapter 7: Better Living in Objectville

- **Inheritance Design:**
  1. Look for shared features.
  2. Extract to a superclass.
  3. Subclass-specific behaviors go in subclass.
- **IS-A vs HAS-A:**
  - Use **IS-A** for inheritance.
  - Use **HAS-A** for object relationships.
- **Polymorphism:**
  - Superclass reference can point to subclass object.
  - Arrays & methods can be polymorphic (e.g., `Animal[]`, `void giveShot(Animal a)`).
- **Overriding:**
  - Method signature must match exactly.
  - Return type must be same or subclass.
  - Cannot reduce access level.
- **Overloading:**

- Same method name, different parameter list.
- Can have different return types & access levels.

✅ Chapter 8: Serious Polymorphism

🔷 Object Creation & Abstract Classes

- **You can't instantiate an abstract class.**
- Abstract classes act as *templates* with partial or no implementation.
- Use them when you want to force subclasses to fill in the details.
- **Abstract methods** have no body; must be implemented by the first concrete subclass.
- A class **must be marked abstract** if it has any abstract method.

🔷 Object Class & Type Safety

- All classes implicitly extend `Object`.
- Don't use `Object` for everything—it breaks **type safety**.
- Java allows only methods declared in the **reference type** to be called.
- Example:

```
Object o = new Ferrari();
o.goFast(); // ❌ Not allowed!
```

🔷 Polymorphism

- Allows objects to be accessed through references of their superclass or interface types.
- Arrays and methods can be polymorphic.
- Compiler checks method calls based on the reference type, not the actual object.

🔷 Interfaces

- Solve the **multiple inheritance** problem.
- You can implement multiple interfaces, regardless of your class's place in the hierarchy.
- Interfaces = roles; Abstract Classes = templates.

🔷 Design Guidelines

- Use a **class** when it passes the IS-A test.
- Use a **subclass** to add/override behaviors.
- Use an **abstract class** when sharing implementation.
- Use an **interface** for roles and cross-tree polymorphism.

✅ Chapter 9: Life and Death of an Object

🔷 Stack vs Heap

- **Stack**: Local variables, method frames.
- **Heap**: All actual objects and instance variables.

🔷 Constructors & Chaining

- **Constructor** is called with `new` and matches the class name.
- If **no constructor is defined**, the compiler adds a default one.
- Every constructor must **call** `super()` (either explicitly or implicitly).
- **Constructor chaining**: Subclass constructor calls superclass constructor first.
- Use `this()` to call another constructor **in the same class** (must be the first line).
- You can call **either** `super()` **or** `this()`, never both.

🔷 Garbage Collection (GC)

An object is **eligible for GC** when no live references point to it:

1. Reference goes out of scope.
2. Reference is reassigned.

3. Reference is set to `null`.

✅ Chapter 10: Numbers Matter

🔷 Wrapper Classes & Boxing

- Each primitive has a **wrapper class**:
  - `int` → `Integer`, `double` → `Double`, `boolean` → `Boolean`, etc.
- Wrapping (manual):

```
Integer iWrap = new Integer(288);
```

- Unwrapping:

```
int unwrapped = iWrap.intValue();
```

🔷 Autoboxing & Unboxing

- Java automatically boxes/unboxes:

```
list.add(x);      // int → Integer
int num = list.get(0); // Integer → int
```

- Works in:
  - Method args/returns
  - Boolean expressions
  - Arithmetic operations
  - Assignments

🔷 Wrapper Utility Methods

- Convert String to primitive:

```
int x = Integer.parseInt("2");
double d = Double.parseDouble("420.24");
```

- Convert primitive to String:

```
String s = "" + d;
String s2 = Double.toString(d);
```

🔷 Number Formatting

- Use `String.format()` for clean numeric output:

```
String s = String.format("%,.2f", 1000000.00);  // 1,000,000.00
```

🔷 Static Concepts

- `static` = shared by all instances.
- Use static for:
  - Utility methods (e.g. `Math.round()`)
  - Constants: `static final int MAX = 100;`
- **Static Initializer Block** runs when class loads:

```
static {
  DOG_CODE = 420;
}
```

Chapter 11 notes on Data Structures and Generics in Java:

🔢 Sorting in Java

- **Default Sorting (Natural Order)**:

Uses `compareTo()` from `Comparable`. Sorts based on Unicode:

Special chars < Numbers < Uppercase < Lowercase.

- **Collections.sort(list)** → Uses natural order (Comparable).

  List.sort(Comparator) → Uses given Comparator.

✅ `Comparable` vs `Comparator`

Comparable	Comparator
Inside the class (e.g., `Song`)	External class (e.g., `ArtistCompare`)
Only ONE sort rule (via `compareTo`)	MANY possible sort rules (via `compare()`)
Affects `Collections.sort(list)`	Used with `List.sort()` or `TreeSet`

👓 Generics

✅ Why Use Generics?

- Compile-time type safety.
- Prevents adding wrong types to collections.
- Example: `ArrayList<String>` only allows Strings.

✅ Generic Class

```
public class ArrayList<E> {
    public boolean add(E o) { ... }
}
```

✅ Generic Method

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

- `<T>` allows the method to be generic.
- Not same as `ArrayList<Animal>` — supports subtype flexibility.

🧠 Lambda + Comparator (Java 8+)

```
songList.sort((s1, s2) → s1.getTitle().compareTo(s2.getTitle()));
```

- No need for separate class.
- Cleaner, shorter, more expressive for one-liners.

📁 Collections Framework Overview

Type	Description
List	Ordered, allows duplicates. Ex: `ArrayList`, `LinkedList`.
Set	No duplicates. Ex: `HashSet`, `TreeSet`.
Map	Key-value pairs. Keys are unique. Ex: `HashMap`, `TreeMap`.

🔐 HashSet & TreeSet

HashSet

- Uses `hashCode()` and `equals()` to check for duplicates.
- Override both for meaningful equality.

TreeSet

- Keeps elements **sorted**.
- Uses either `compareTo()` or a `Comparator`.

🗺️ Maps

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Bert", 343);
scores.get("Bert"); // 343
```

- Not a subtype of Collection but part of the Collection Framework.

- Uses key-value pairs.
- No duplicate keys allowed.

✨ Convenience Factory Methods (Immutable Collections)

- `List.of()`, `Set.of()`, `Map.of()` — Immutable!

```
List<String> strings = List.of("A", "B", "C");
Map<String, Integer> scores = Map.of("A", 1, "B", 2);
```

🧬 Generics with Wildcards

✅ Why `List<? extends Animal>`?

- Accepts `List<Dog>`, `List<Cat>` etc.
- Can't add to it (read-only in this context).

```
public void takeAnimals(List<? extends Animal> animals) {}
```

✅ Generic Return + Parameter

```
public <T extends Animal> List<T> takeAnimals(List<T> list)
```

- Ensures type safety both ways (input and return).

✅ Chapter 12: Lambdas and Streams – What, Not How

☑️ Core Concepts

- **Streams** = pipeline of data operations (`source → intermediate ops → terminal op`)

- **Lambdas** = anonymous functions that implement a *functional interface* (1 abstract method)

🔁 Common List Processing

```
list.forEach(item → System.out.println(item)); // lambda passed to forEach
```

🏞️ Streams API

- Created using `.stream()` on a collection
- **Intermediate operations** (e.g., `map`, `filter`, `sorted`, `limit`) → return new Stream (lazy)
- **Terminal operations** (e.g., `collect`, `count`, `forEach`) → trigger execution (eager)

🧱 Stream Pipeline Structure

```
List<String> result = strings.stream()
                    .sorted()
                    .limit(4)
                    .collect(Collectors.toList());
```

⚠️ Stream Rules

1. Need a **stream()** and a **terminal operation**
2. **Streams are single-use** – you can't reuse them after a terminal op
3. Don’t modify the original collection while streaming

🧺 Collecting

- `Collectors.toList()`, `toSet()`, `toMap()`, `toUnmodifiableList()` (Java 10+)
- `Collectors.joining(", ")` for concatenated strings

🔄 map() and filter()

```
songs.stream()
    .filter(song → song.getGenre().contains("Rock"))
    .map(Song::getGenre)
    .collect(Collectors.toList());
```

💡 Method References

- Instead of `x → x.getY()`, use `ClassName::getY`

```
Comparator<Song> byYear = Comparator.comparingInt(Song::getYear);
```

🔍 Matching & Finding

```
anyMatch(p), allMatch(p), noneMatch(p)
findFirst(), findAny(), max(), min(), reduce()
```

❓ Optional

- Wrapper for potentially missing values

```
Optional<Song> maybeSong = stream.findFirst();
maybeSong.isPresent()
```

✅ Chapter 13: Risky Behavior

🎵 JavaSound API

- MIDI is instructions, not actual sound
- Requires a `Sequencer` to play music

```
Sequencer s = MidiSystem.getSequencer(); // throws MidiUnavailableException
```

⚠️ Exception Basics

- **Checked exceptions** must be declared or caught (`IOException`, etc.)
- **Unchecked exceptions** (RuntimeException and subclasses) don't need to be caught

🔁 Try/Catch/Finally Flow

- `try` → if error → `catch` → then `finally`
- `finally` always runs (even after return in try/catch)

🎯 Multiple Catches & Polymorphism

- Catch specific exceptions before general ones
- Catch blocks must go from most specific to most general

```
try {
  ...
} catch (TeeShirtException t) {
} catch (ClothingException c) {
}
```

🙈 Ducking Exceptions

- Declare with `throws` to pass handling up the call stack

```
public void risky() throws SomeException { ... }
```

- If no one handles it, JVM crashes the program

📜 Exception Rules

1. `try` must be followed by `catch` or `finally`
2. Can't put code between `try` and `catch`
3. `finally` always runs
4. No catch/finally = compilation error

🔄 Lambda Quick Recap

```
() → System.out.println("Hi!")              // Runnable
str → System.out.println(str)              // Consumer<T>
(a, b) → a.compareToIgnoreCase(b)          // Comparator<T>
```

- Lambdas implement *functional interfaces* (SAMs)
- Syntax:
  - `(params) → expression`
  - `(params) → { statements; return value; }`

🔷 Chapter 14: A Very Graphic Story

✅ Basics of Swing GUI

- `JFrame`: Main window where all components (buttons, labels, etc.) are added.
- Use `frame.getContentPane().add()` to add components—not directly to `JFrame`.

✅ Event Handling in Java

- **Event Source**: Button (generates an event like a click).
- **Listener Interface**: You (implementing ActionListener).
- **Event Object**: Holds event data (`ActionEvent`, `MouseEvent`, etc.).

ActionListener Setup:

1. `implements ActionListener`
2. `button.addActionListener(this)`
3. Define `public void actionPerformed(ActionEvent e)`

✅ Graphics with Swing

- Override `paintComponent(Graphics g)` in a `JPanel` subclass.
- Use `Graphics2D` for advanced drawing:

```
Graphics2D g2d = (Graphics2D) g;
```

Drawing Examples:

- `g.setColor(Color.BLUE);`
- `g.fillOval(x, y, width, height);`
- `g.drawImage(image, x, y, this);`

Gradient Example:

- `GradientPaint` lets you blend two colors.
- Call `repaint()` to trigger a redraw of components.

✅ Handling Multiple Buttons

- **DON'T**: Implement multiple `actionPerformed()` methods.
- **DO**:
  - Use **inner classes** for each listener (best OO approach).
  - Inner class has access to outer class variables.
  - Lambdas also work:
    `button.addActionListener(e -> label.setText("Ouch!"));`

🔷 Chapter 15: Work on Your Swing

✅ Component & Container

- Everything visible is a **component**.
- Most components can **contain other components** (are also containers).

✅ Layout Managers

1. **BorderLayout** (default for JFrame):
    - 5 regions: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`
    - Center gets leftover space.
    - Use: `add(BorderLayout.EAST, myPanel);`
2. **FlowLayout** (default for JPanel):
    - Left to right, wraps when out of space.
    - Honors preferred size.
3. **BoxLayout**:
    - Stack components vertically (`Y_AXIS`) or horizontally (`X_AXIS`).
    - Honors preferred size.
    - Use: `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));`

✅ Sizing Rules

- Layout managers **ask for preferred size**, but may ignore it.
- `NORTH/SOUTH`: Keep preferred height, stretch width.
- `EAST/WEST`: Keep preferred width, stretch height.

✅ Important Code Patterns

Registering a Listener:

```
button.addActionListener(new MyListener());
```

Graphics 101:

```
public void paintComponent(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;
  g2.setPaint(new GradientPaint(70, 70, Color.RED, 150, 150, Color.BLUE));
  g2.fillOval(70, 70, 100, 100);
}
```

✅ **Pro Tip:**

Don't call `paintComponent()` directly—call `repaint()` instead. JVM handles the rest.

Chapter 16: Saving Objects (and Text) in Java

1. Saving State in Java

- Use **serialization** to save Java objects for use *only* by your program.
- Use **plain text files with delimiters** (comma, tab) if other programs need to read the data.
- Data moves via **streams**:
  - **Connection streams** (e.g., `FileOutputStream`) connect to sources/destinations.
  - **Chain streams** (e.g., `ObjectOutputStream`) add functionality and wrap connection streams.

2. Serialization Basics

- Objects saved as bytes, storing instance variables and class info.
- **Entire object graph** is saved (all referenced objects).
- To serialize, class must implement `Serializable` (a marker interface).
- Use `transient` keyword to skip variables during serialization.
- Serialization is **all or nothing**—any non-serializable object in the graph causes failure.
- Serialize using `ObjectOutputStream.writeObject()`.
- Deserialize with `ObjectInputStream.readObject()` and cast back to original type.
- Static variables are **not** serialized.
- Use `serialVersionUID` to manage class version compatibility.

3. Deserialization

- JVM recreates objects without calling their constructors.

- Non-serializable superclass constructors *do* run.
- All classes in the object graph must be present at runtime.
- Read objects in the same order they were written.

4. File I/O for Text

- Use `FileWriter` + `BufferedWriter` to write text files efficiently.
- Use `FileReader` + `BufferedReader` to read text files line by line.
- Use `String.split()` to parse delimited text files.

5. Java NIO.2 Package

- `java.nio.file` provides modern file and directory manipulation.
- Use `Path` (via `Paths.get()`) to locate files/directories.
- Use `Files` class for reading/writing text, creating files, walking directories.
- Example to create a writer: `BufferedWriter writer = Files.newBufferedWriter(myPath);`

6. Try-With-Resources (TWR)

- Automatically closes resources (files, streams) when done or if an exception occurs.
- Only works with classes implementing `AutoCloseable`.
- Multiple resources declared close in reverse order.

Quick Code Snippets:

Serialization:

```
ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream
("Game.ser"));
os.writeObject(myObject);
os.close();
```

Deserialization:

```
ObjectInputStream is = new ObjectInputStream(new FileInputStream("Ga
me.ser"));
MyClass obj = (MyClass) is.readObject();
is.close();
```

Writing Text File:

```
BufferedWriter writer = new BufferedWriter(new FileWriter("file.txt"));
writer.write("Hello, world!");
writer.close();
```

Reading Text File:

```
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
reader.close();
```

Try-With-Resources Example:

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter("file.txt")))
{
    writer.write("Hello");
} catch (IOException e) {
    e.printStackTrace();
}
```

Chapter 17 — Making a Connection

- **Channels & Sockets**:
  - `Socket` = blocking I/O, simpler, good for few clients.

- `SocketChannel` = non-blocking (NIO), scalable, works with `Selector`, better for many clients.

- **Client-Server Basics**:
  - Client needs server IP and port to connect.
  - Ports identify applications (0-1023 reserved for well-known services).
  - Example:

```
SocketChannel socketChannel = SocketChannel.open(new InetSo
cketAddress("127.0.0.1", 4200));
Reader reader = Channels.newReader(socketChannel, StandardCh
arsets.UTF_8);
BufferedReader bufferedReader = new BufferedReader(reader);
String message = bufferedReader.readLine();
```

- **Reading/Writing Data**:
  - Use `BufferedReader` for reading from server.
  - Use `PrintWriter` for writing to server.
- **ServerSocketChannel** listens for connections; `accept()` blocks until client connects, returns a `SocketChannel` for client.
- **Limitation**: Single-threaded server handles one client at a time; to serve multiple clients simultaneously, use **multithreading**—start a new thread per client connection.

Multithreading in Java

- **Thread vs thread**:
  - `Thread` (capital T) is Java class representing a thread of execution.
  - Every Java app starts with main thread.
- **Runnable interface** has a single `run()` method — job for the thread.
- **Starting a thread**:

```
Runnable job = new MyRunnable();
Thread thread = new Thread(job);
thread.start(); // NOT run()
```

- **Better approach**: Use **ExecutorService** from `Executors` class to manage threads and thread pools, avoid manual thread management.
- **Thread states**: NEW → RUNNABLE → RUNNING → (Blocked/Sleeping/Waiting).
- **Sleeping threads**:
  - `Thread.sleep(milliseconds)` pauses thread temporarily (needs try/catch).
  - Doesn't guarantee immediate running after waking up; thread scheduler decides.
- **Synchronization tools**:
  - `CountDownLatch` for waiting on multiple threads/events.
- **ExecutorService shutdown**:
  - Use `shutdown()`, then `awaitTermination(timeout)`, then `shutdownNow()` as last resort.

Chapter 18 — Dealing with Concurrency Issues

- **Concurrency problems arise when multiple threads access/modify shared data simultaneously.**
- **Race condition / Lost update**:

  Non-atomic operations (like check-then-act) cause inconsistent results.
- **Ryan and Monica Problem**: Two threads withdraw money simultaneously causing overdraft because check and spend aren’t atomic.
- **Solution: Synchronization**
  - Use `synchronized` keyword on method or block to ensure atomicity (only one thread holds lock on object).
  - Example:

```
synchronized(account) {
  if(account.getBalance() >= amount) {
    account.spend(amount);
  }
}
```

- **Atomic variables** (`AtomicInteger`, `AtomicLong`) provide lock-free thread-safe operations using compare-and-swap (CAS).
- **Deadlock**: Occurs when two threads hold locks the other needs, causing indefinite waiting.
- **Immutable Objects**:
  - Make classes final with final fields and no setters.
  - Thread-safe by design; no synchronization needed.
- **Thread-safe Collections**:
  - Use `CopyOnWriteArrayList` for read-heavy, write-light scenarios to avoid `ConcurrentModificationException`.
- **Summary of synchronization key points**:
  - Each object has one lock.
  - Only one thread can hold a lock at a time.
  - Synchronizing a method or block locks the object's key.
  - Threads waiting for a lock enter a waiting state.
- **Optimistic locking (CAS)** allows multiple threads to attempt updates without blocking but may require retries if conflicts occur.

JShell REPL

- **What it is:** A REPL (Read-Eval-Print Loop) introduced in JDK 9.
- **Usage:** Allows running Java code snippets interactively.
- **Key Commands:**
  - `/vars`: Lists all variables.

- `/exit`: Exits the REPL.

Packages

- **Purpose:** Prevent class name conflicts using reverse domain naming (e.g., `com.headfirstjava`).
- **How it works:**
  - Requires matching directory structure.
  - Use `package` statement in the source file.
- **Compiling:**
  - Use `javac -d` flag to specify the destination directory for the package.

Immutability in Strings & Wrappers

- **Strings & Wrapper Classes:** Immutable by design.
  - String manipulation creates new objects (e.g., `new String()`).
  - Use `StringBuilder` for frequent modifications.
- **Wrapper Classes:** (e.g., `Integer`, `Double`)
  - Cannot modify their values after instantiation, but can refer to new objects.

Varargs

- **Definition:** Allows methods to accept a variable number of arguments.
- **Syntax:** `Object... items`
- **Rules:**
  - Only one varargs parameter per method.
  - Must be the last parameter in the method signature.

```
void printAll(Object... items) {
  for (Object item : items) {
    System.out.println(item);
```

```
 }
}
printAll("A", 42, true);  // Accepts any number of arguments.
```

Annotations

- **Definition:** Provide metadata or behavior for code.
- **Common Examples:**
  - `@Test` (JUnit).
  - `@SpringBootApplication` (Spring).
- **Usage:** Applied to classes, methods, or variables.
- **Can Have Elements:** (e.g., `@Override`).

Lambdas & Maps

- **Java 8 Features:** Lambdas simplify code with functional-style operations.
- **Map Methods:**
  - `computeIfAbsent`: Adds a new value if the key is absent.
  - `computeIfPresent`: Updates the value if the key exists.

```
Map<String, Integer> scores = new HashMap<>();
scores.computeIfAbsent("Alice", k -> 100);  // Adds "Alice=100".
scores.computeIfPresent("Alice", (k, v) -> v + 10);  // Updates to 110.
```

Parallel Streams

- **What it is:** Enables parallel processing of streams for performance gains.
- **When to Use:** Only beneficial for large datasets or complex operations.
- **How to Start Parallel Processing:**
  - `parallelStream()`

- `.stream().parallel()`

```
List<Song> songs = getSongs();
Stream<Song> par = songs.parallelStream();
```

Enums

- **Definition:** Special classes representing fixed sets of constants.
- **Benefits:** Improve code readability, type safety, and can be used in `if` / `switch`.

```
public enum BandMember { KEVIN, BOB, STUART }
BandMember member = BandMember.KEVIN;

switch (member) {
  case KEVIN → System.out.println("Guitar!");
  case BOB   → System.out.println("Bass!");
}
```

Local Variable Type Inference (`var`)

- **Java 10 Feature:** Allows the compiler to infer the type of local variables.
- **Key Limitations:**
  - Cannot use `var` without initialization.
  - Cannot change type once inferred.

```
var list = new ArrayList<String>();  // Inferred as ArrayList<String>.
var name = "Avish";  // Inferred as String.
```

Records

- **What it is:** A special class for immutable data. Introduced in Java 16.
- **Features:** Auto-generates constructors, `equals()`, `hashCode()`, and `toString()`.

- **Custom Validation:** Support compact constructors with validation logic.

```
public record Point(int x, int y) {}  // Auto-generates constructor, accessors et
c.
Point p = new Point(1, 2);
System.out.println(p.x());  // 1 (no "get" prefix).
```

- **Compact Constructor:**

```
public record Point(int x, int y) {
  public Point {
    if (x < 0) throw new IllegalArgumentException();
  } // no params needed
}
```