Java Programming (DI03032021)

Complete Lecture Deck – All 45 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Table of Contents I

Table of Contents II

Table of Contents III

Table of Contents IV

Table of Contents V

Table of Contents VI

Table of Contents VII

Table of Contents VIII

Course Overview & Today's Agenda

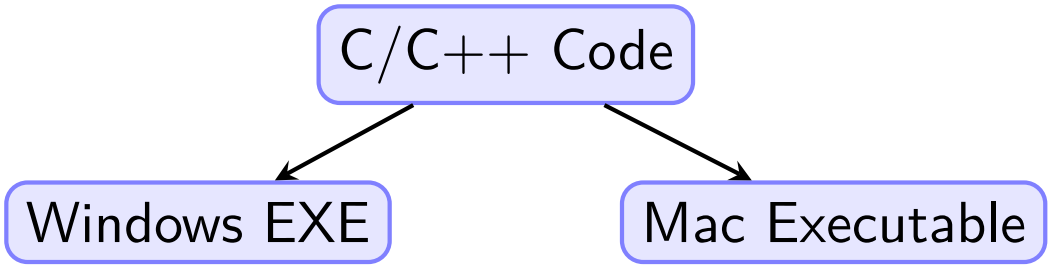
- Course spans 5 units across 15 weeks (45 lectures, 3 lectures/week)
- Unit 1: Java Fundamentals (9) - Unit 2: OOP (9) - Unit 3: Inheritance/Interfaces/Packages (9)
- Unit 4: Exceptions & Multithreading (9) - Unit 5: Collections & File Handling (9)
- Today: History of Java, its key features, and the JVM architecture behind 'write once, run anywhere'

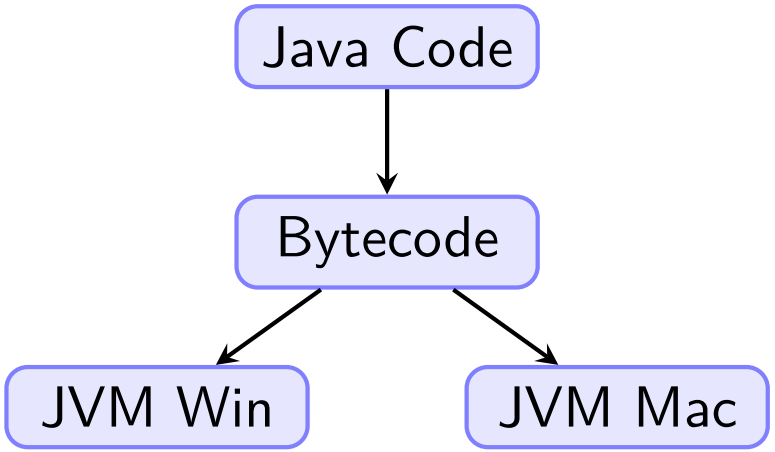
A Brief History of Java

- Created by James Gosling and team at Sun Microsystems, starting in 1991
- Originally named 'Oak' (after a tree outside Gosling's office), renamed 'Java' in 1995
- Designed first for interactive TV/set-top boxes, then repurposed for the World Wide Web as applets
- Now owned and maintained by Oracle Corporation since 2010; current LTS releases include Java 17 and Java 21

Why a New Language? The Design Motivation

- C/C++ programs were fast but platform-dependent and unsafe
- Goal: 'Write Once, Run Anywhere' (WORA)





Key Features of Java - Part 1

Simple
Clean syntax derived from C/C++ without complex pointers

Object-Oriented
Everything except primitives is modeled as objects

Platform-Independent
Bytecode runs identically on any OS through the JVM

Secure
No explicit pointers, bytecode verification, sandboxing

Key Features of Java - Part 2

Robust
Strong checking; automatic garbage collection

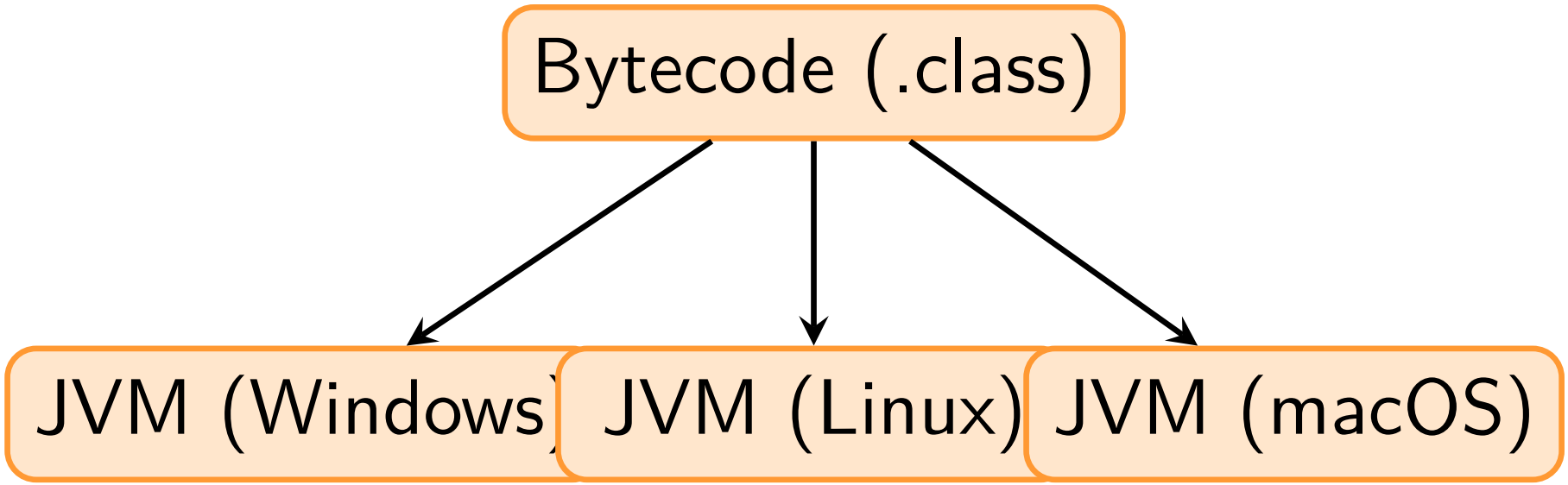
Multithreaded
Built-in support for concurrent execution

High Performance
JIT compiled to native machine code at run time

Distributed & Portable
Rich networking and fixed primitive sizes

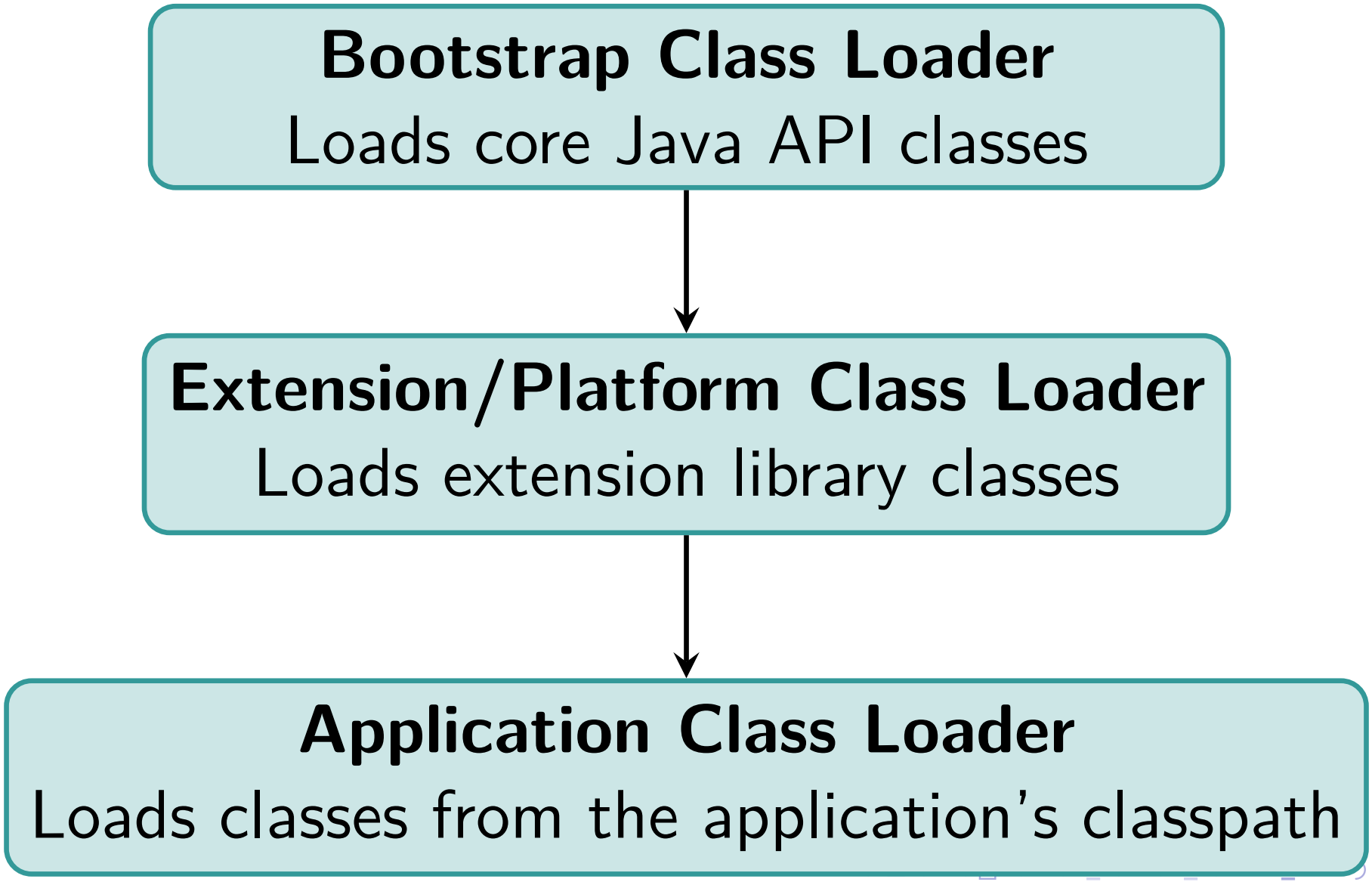
What is the JVM?

- JVM = Java Virtual Machine, an abstract machine that runs Java bytecode
- The one layer of abstraction behind 'write once, run anywhere'



JVM Architecture: Class Loader Subsystem

- Loads compiled .class files into memory when a program starts or a class is first referenced



JVM Architecture: Runtime Data Areas

- Stores data required during execution

Method Area
Class-level data (shared)

Heap
Objects created with 'new' (shared, GC)

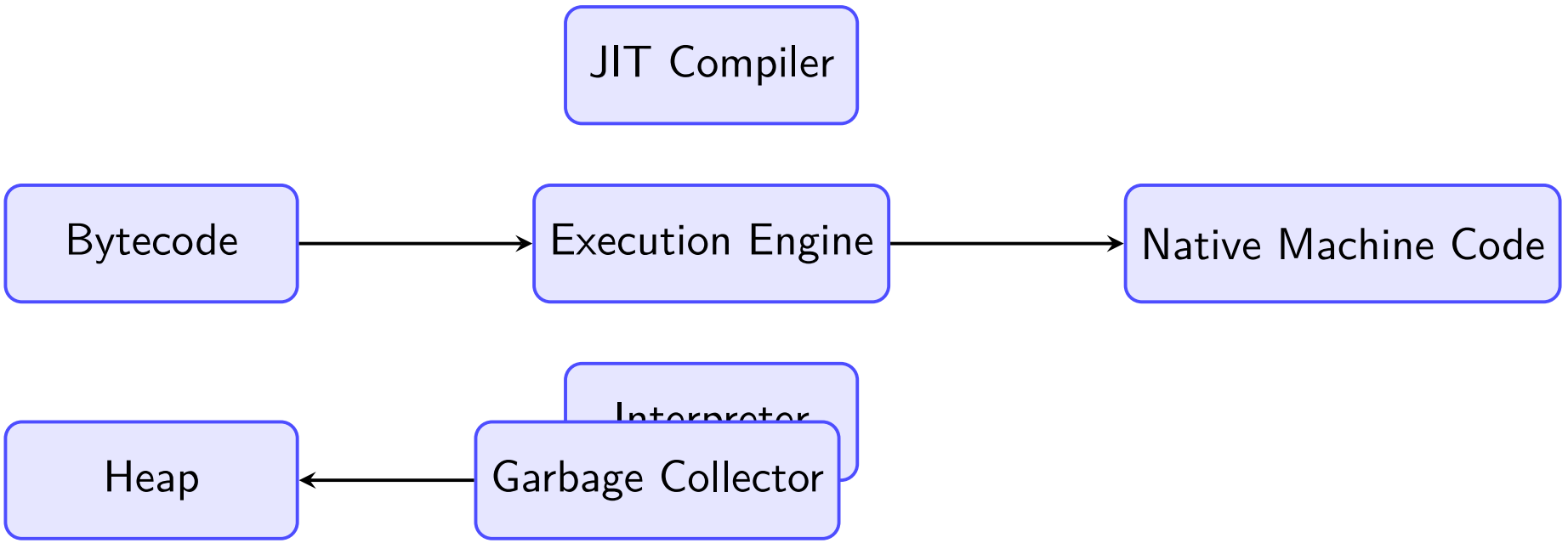
Stack
Method call frames, local variables (per thread)

PC Register
Current instruction (per thread)

Native Method Stack
Native code calls

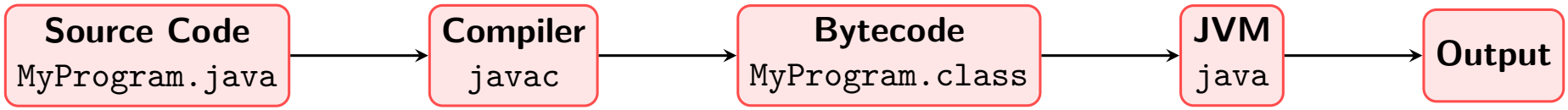
JVM Architecture: Execution Engine

- **Interpreter:** reads and executes bytecode instructions one at a time
- **JIT (Just-In-Time) Compiler:** compiles frequently used bytecode into native machine code
- **Garbage Collector:** automatically reclaims heap memory occupied by objects no longer in use



Compilation and Execution Flow: The Big Picture

- This exact flow is demonstrated hands-on in Lecture 4 and Lecture 5



Where Java is Used Today

- Android app development (Android Runtime evolved from JVM concepts)
- Enterprise backend systems - banking, insurance, and e-commerce servers (Java EE / Spring)
- Big data platforms such as Hadoop and Spark are built largely in Java
- Embedded systems and set-top boxes - Java's original target domain

Quick Check: Test Your Understanding

- Q: What does JVM stand for, and what does it actually execute?
- Q: Name two features that make Java 'robust'
- Q: Why does 'write once, run anywhere' work only because of the JVM?

Key Takeaways

- Java was created by Sun Microsystems (1991, originally 'Oak') and is now maintained by Oracle
- Java's key features include being simple, object-oriented, platform-independent, robust, secure, and multithreaded
- The JVM executes platform-independent bytecode - the real mechanism behind 'write once, run anywhere'
- The JVM has three core parts: Class Loader Subsystem, Runtime Data Areas, and Execution Engine

Next Lecture Preview

- Lecture 2 completes topic 1.1 by covering Java applications and how platform independence works in practice across different operating systems.

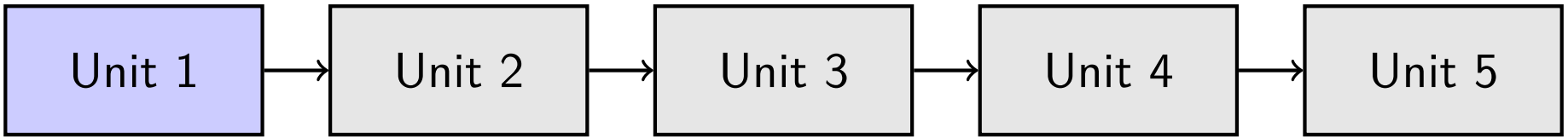
Questions?

Recap: The JVM and Bytecode

- JVM = Java Virtual Machine, executes platform-independent bytecode (.class files)
- Class Loader Subsystem, Runtime Data Areas, and Execution Engine are the three JVM components
- Bytecode, not source code, is what makes Java portable across operating systems

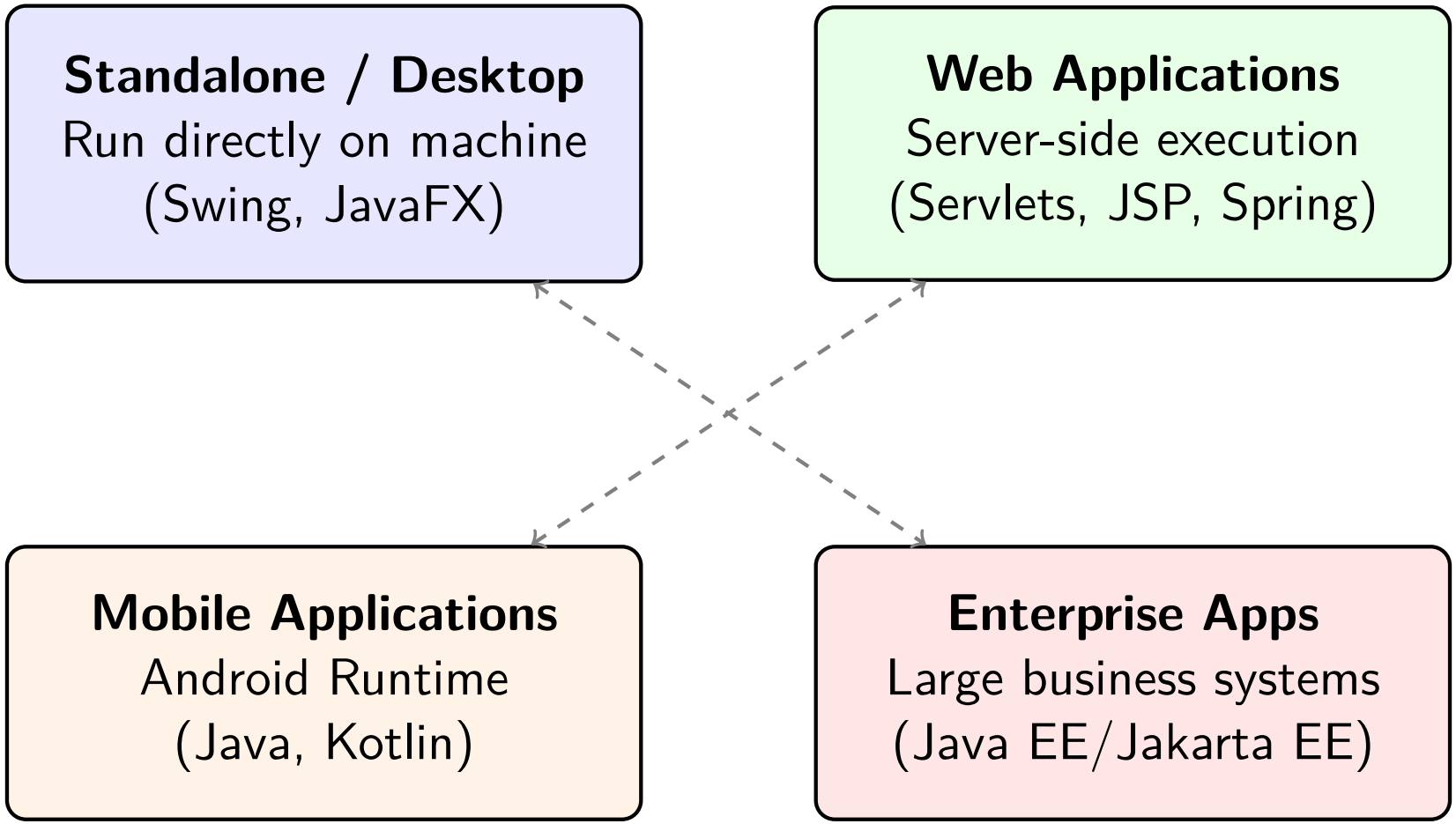
Course Overview & Today's Agenda

- Recap: JVM architecture and why bytecode enables portability
- Types of Java applications (standalone, web, enterprise, mobile, embedded)
- Platform independence in depth: bytecode vs native compilation vs pure interpretation
- Wrap-up of topic 1.1 before moving to JDK/JRE/JVM setup in Lecture 3



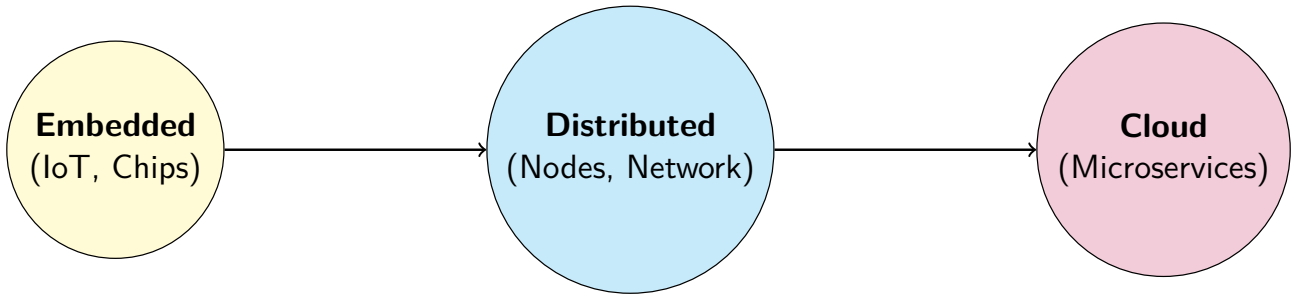
Categories of Java Applications

- Standalone/Desktop, Web, Mobile, and Enterprise apps



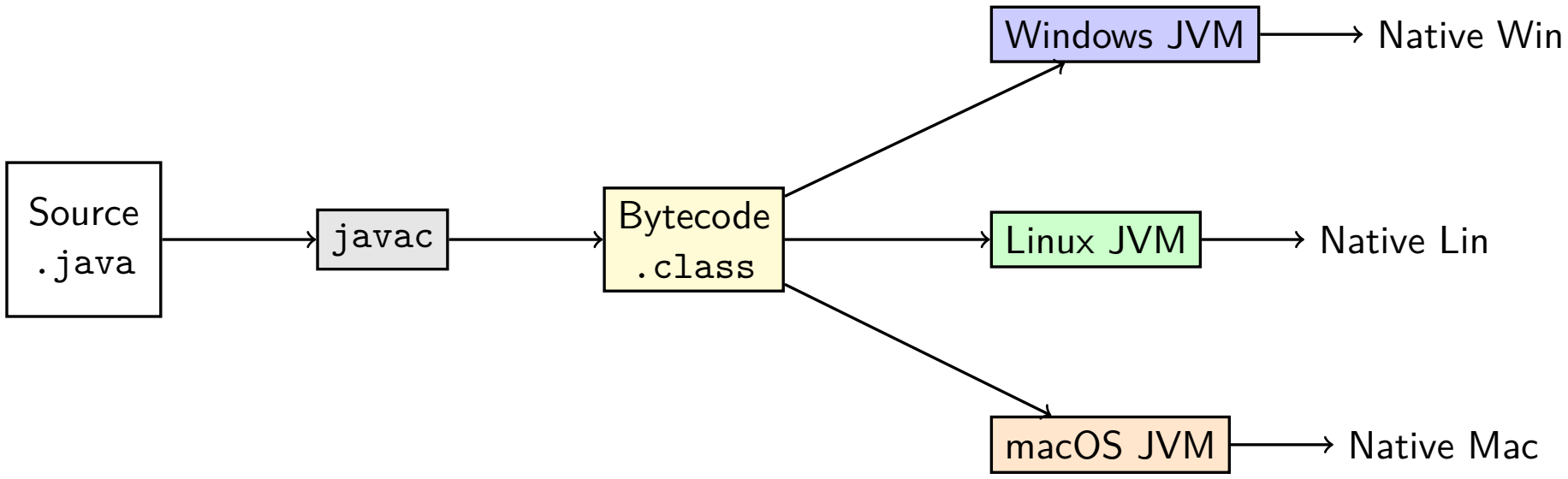
Embedded and Distributed Java Applications

- Embedded systems: smart cards, set-top boxes, and IoT devices (Java's original 1991 target domain)
- Distributed applications: Java's networking APIs (java.net, RMI) support client-server and distributed computing
- Cloud-native applications: many microservices frameworks (Spring Boot, Micronaut) are Java-based
- This breadth of use cases is only possible because the same bytecode runs everywhere



Platform Independence: The Precise Mechanism

- Source code is compiled ONCE by javac into bytecode (.class)
- Bytecode is translated into native instructions by OS-specific JVM



Compiled vs Interpreted vs Java's Hybrid Model

- This hybrid gives Java both portability AND near-native performance

Model	Mechanism	Pros / Cons
Purely Compiled	Source → Machine Code (C/C++)	Very fast, but not portable
Purely Interpreted	Source executed line-by-line (Python)	Highly portable, but slower
Java Hybrid	Source → Bytecode → JIT Machine Code	Portable AND fast

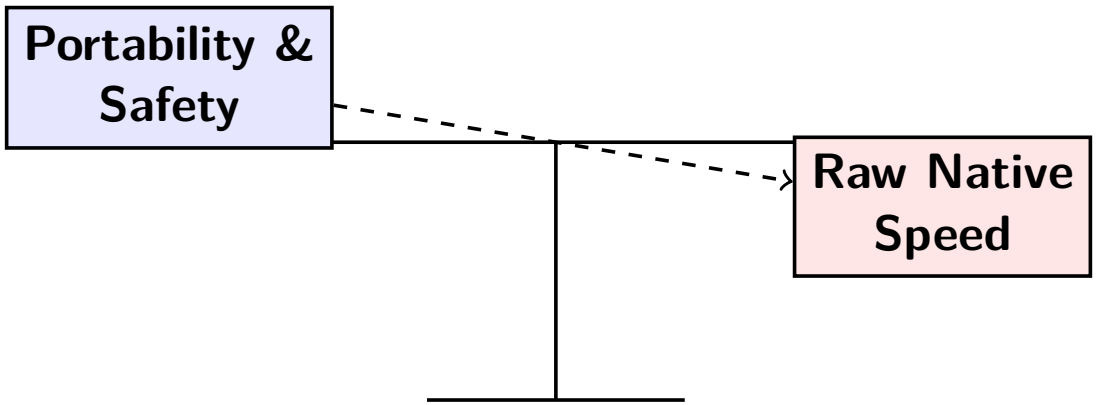
What Platform Independence Does NOT Mean

- Platform independence applies to the compiled bytecode of pure Java code

Myth (What it is NOT)	**Fact (What it IS)**
No JVM is needed	A compatible JVM MUST be installed
Identical performance everywhere	JIT and hardware still affect speed
Native libraries work everywhere	Native libs are OS-specific

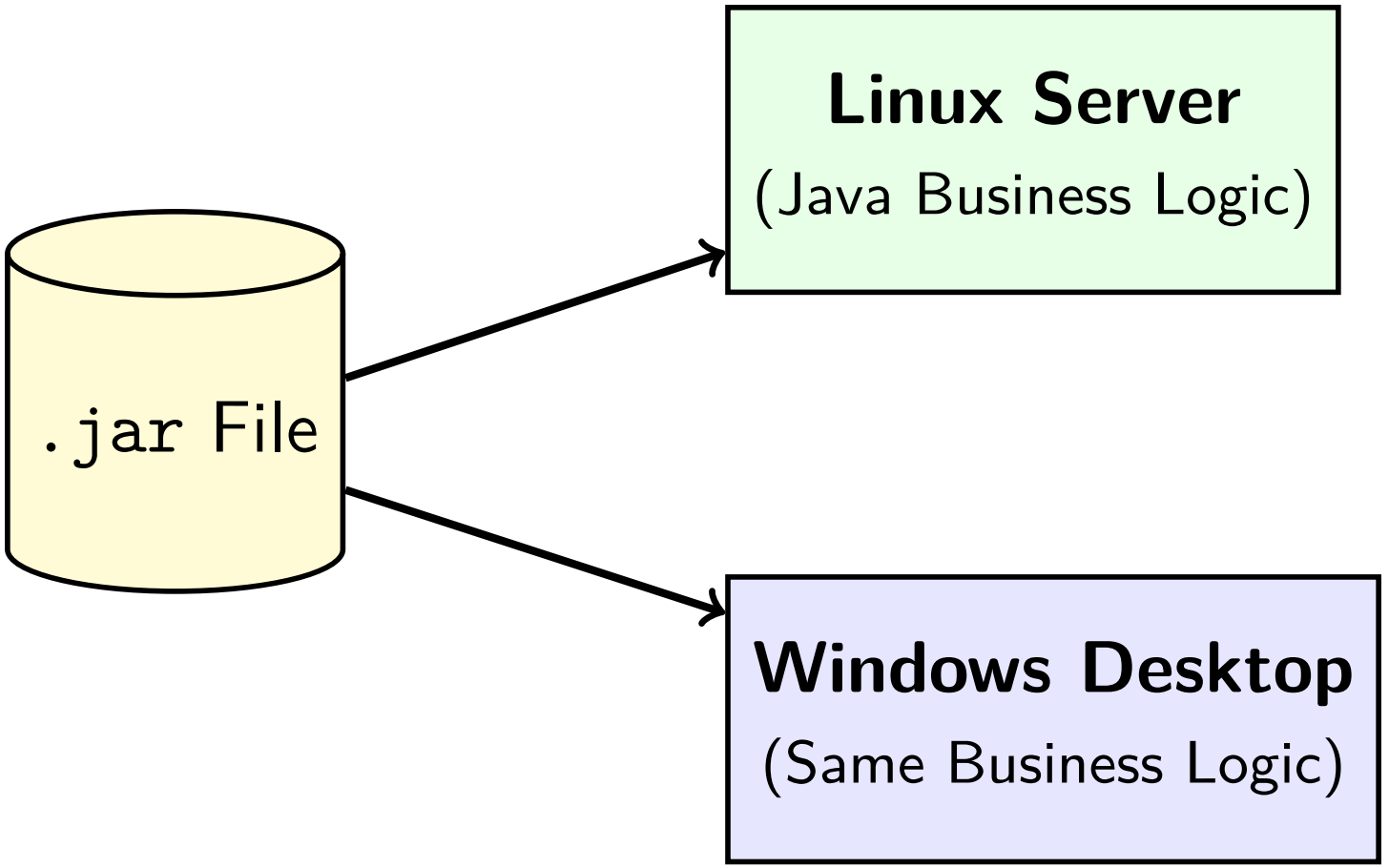
The Trade-off: Portability vs Raw Speed

- Historically, Java was seen as slower due to interpretation layer
- Modern JIT compilation closed much of this gap
- For most business workloads, this trade-off favors Java's model



Real-World Example: One Codebase, Many Devices

- A banking application's Java module runs on Linux server and Windows workstation
- Eliminates the 'works on my machine' problem



Unit 1 Topic 1.1: Complete

- 1.1 is now fully covered: Java's history and features, JVM architecture, applications, and platform independence
- Next stop: getting Java actually installed and configured on a machine (JDK/JRE/JVM) in Lecture 3

Topic 1.1 Complete

Quick Check: Test Your Understanding

Questions

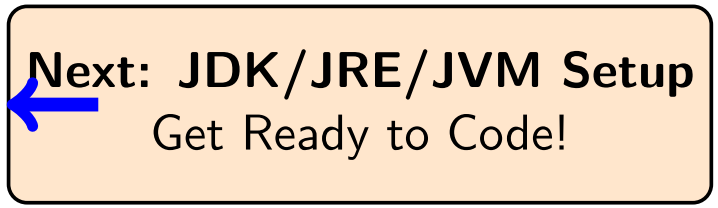
- Q: Why can the same .class file run on Windows and Linux without recompilation?
- Q: Name two categories of real-world Java applications
- Q: Is Java purely compiled, purely interpreted, or a hybrid? Explain in one sentence

Key Takeaways

- Java applications span desktop, web, mobile (Android), enterprise, embedded, and cloud-native systems
- Platform independence works because javac compiles source into portable bytecode ONCE; each OS's JVM then executes that same bytecode natively
- Java uses a hybrid compiled+interpreted model: bytecode compilation for portability, JIT compilation for speed
- A compatible JVM must still be installed on every target machine - portability applies to the bytecode, not to 'needing nothing at all'

Next Lecture Preview

- Lecture 3 gets hands-on: installing and configuring the JDK/JRE, understanding their relationship to the JVM, and setting up the Java development environment and tools.



Questions?

Recap: Platform Independence

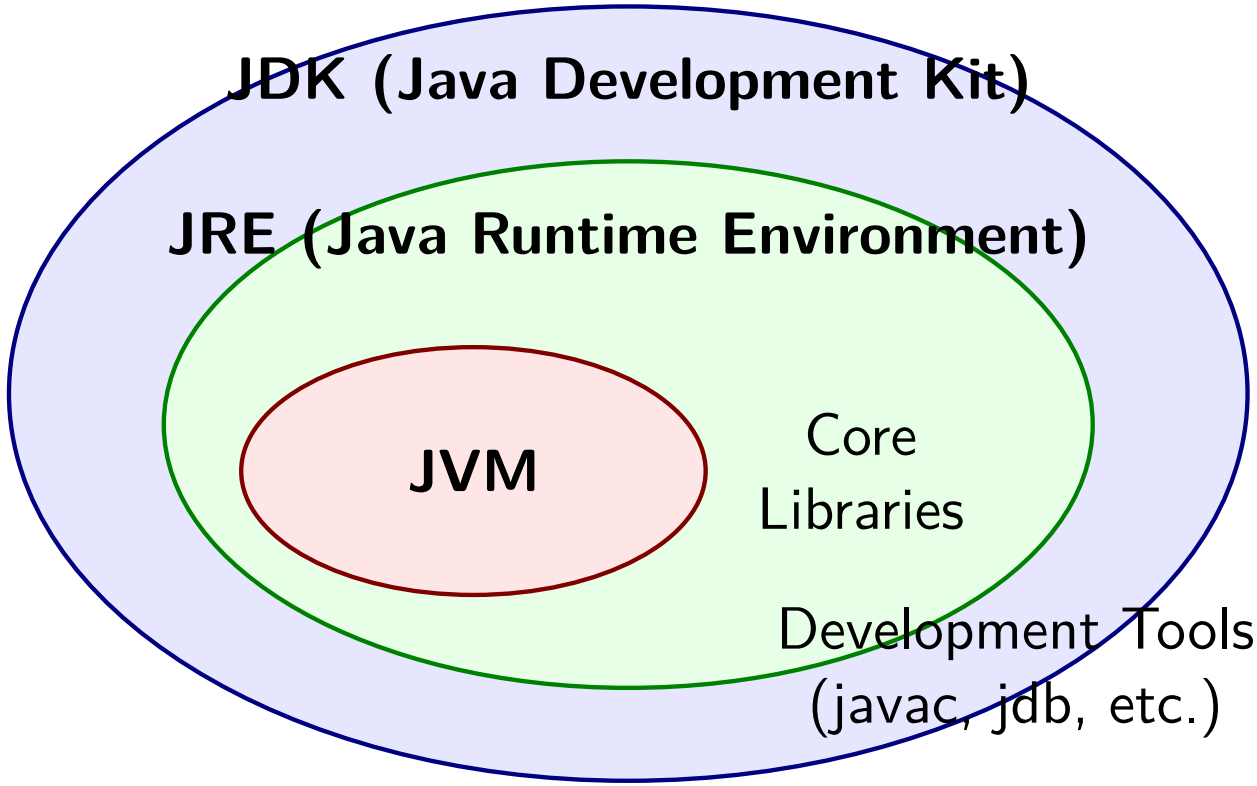
- javac compiles .java source into portable bytecode (.class) exactly once
- Each operating system's own JVM executes that identical bytecode natively
- Java applications span desktop, web, mobile, enterprise, and embedded systems

Course Overview & Today's Agenda

- JDK vs JRE vs JVM: what each one actually contains
- Key tools bundled inside the JDK
- Installing and configuring the Java environment (JAVA_HOME, PATH)
- Common Java IDEs and development tools used in industry

JDK, JRE, and JVM: The Three Layers

- JVM (Java Virtual Machine): executes bytecode - innermost layer
- JRE (Java Runtime Environment): JVM + core class libraries needed to RUN
- JDK (Java Development Kit): JRE + development tools needed to WRITE and COMPILE



JDK vs JRE: Who Needs What?

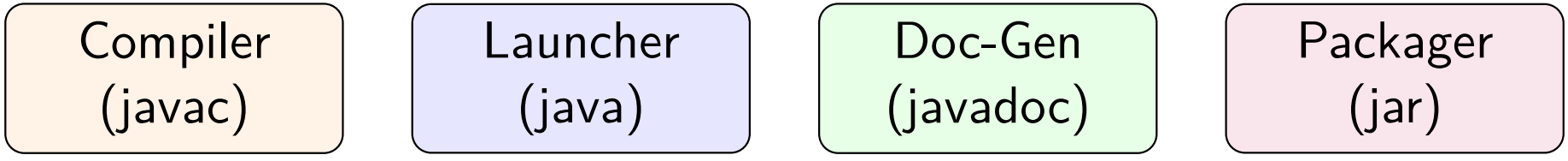
JDK (For Developers)	JRE (For End Users)
A developer writing and compiling Java code needs the full JDK	An end user who only RUNS a Java application needs just the JRE
Contains JVM, Libraries + Development tools	Contains JVM and Libraries only

Exam tip

JDK = 'for developers', JRE = 'for running programs', JVM = 'the engine that executes bytecode'. Modern JDK distributions (Java 11+) bundle everything together.

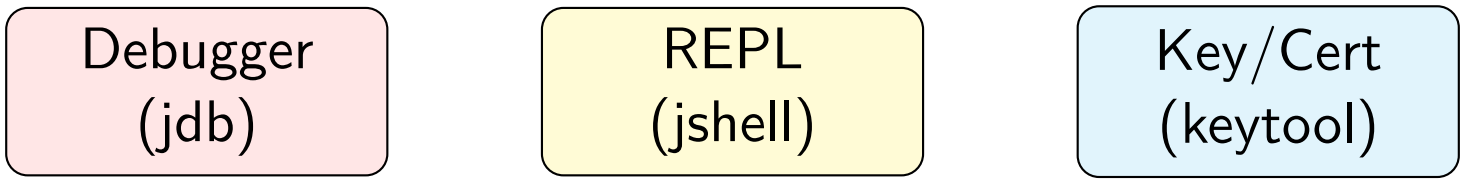
Key Tools Bundled Inside the JDK

- **javac** - the compiler; translates .java source files into .class bytecode files
- **java** - the launcher; starts the JVM and runs a compiled program's bytecode
- **javadoc** - generates HTML API documentation from specially formatted source comments
- **jar** - packages multiple .class files (and resources) into a single distributable .jar archive



More JDK Tools You Will Meet Later

- **jdb** - the Java debugger, for stepping through running code and inspecting variables
- **jshell** - an interactive Java shell (REPL) for quickly trying out snippets
- **keytool** - manages cryptographic keys and certificates for secure applications



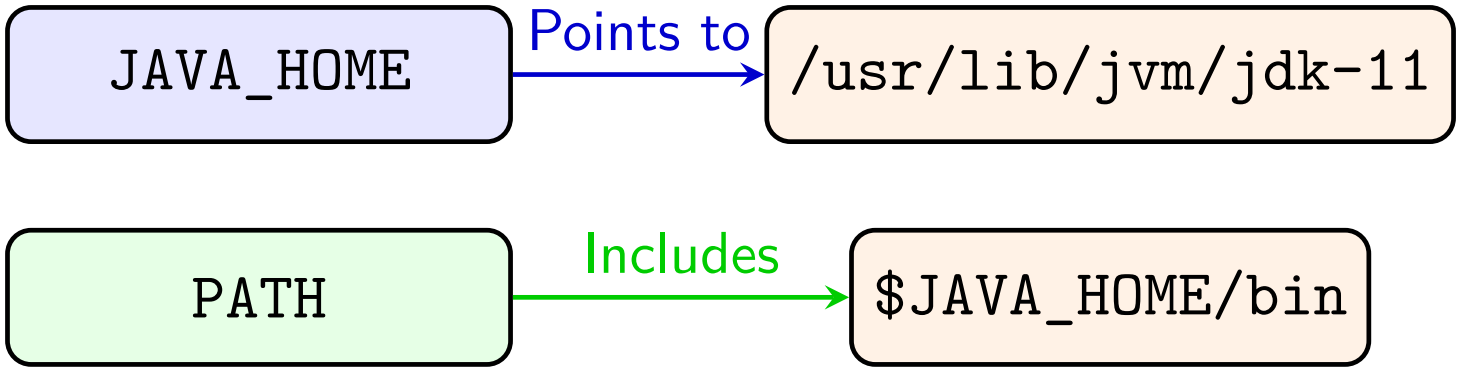
Installing the JDK

- Download a JDK distribution (e.g., Oracle JDK, or Eclipse Temurin) for your OS
- Run the installer, which places the JDK's `bin/` folder (javac, java, etc.) on disk
- Verify installation by running `java -version` and `javac -version` from a terminal
- Course lab resources use JDK version 11 or above

```
$ java -version
openjdk version "11.0.12" 2021-07-20
OpenJDK Runtime Environment...
```

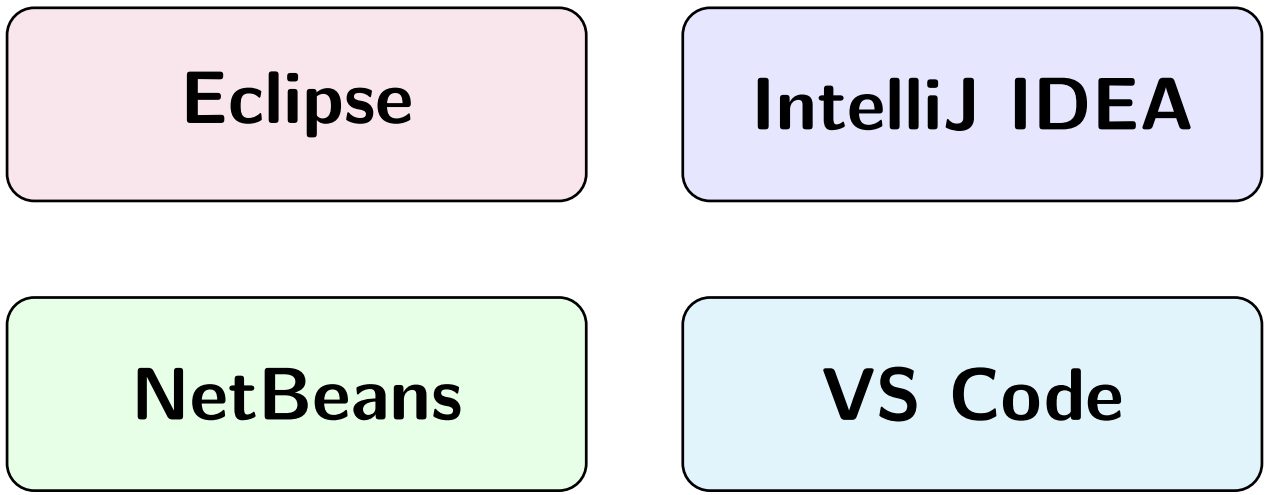
Configuring Environment Variables

- **JAVA_HOME**: points to the JDK's installation directory
- **PATH**: must include the JDK's `bin/` folder so `javac` and `java` work from anywhere
- Without PATH configuration, terminal reports 'javac is not recognized'
- Windows: System Properties > Environment Variables; Linux/macOS: .bashrc/.zshrc



Common Java IDEs and Development Tools

- **Eclipse**: free, widely used open-source IDE with strong plugin ecosystem
- **IntelliJ IDEA**: modern IDE with excellent code assistance
- **NetBeans**: free, mature open-source IDE with built-in GUI builders
- **Visual Studio Code**: lightweight editor with Java extensions



Text Editor vs IDE: What's the Difference?

Text Editor + Terminal	IDE
A plain text editor (Notepad, vi) only edits text	Bundles editing, compiling, running, and debugging in one tool
Must run javac/java manually from the terminal	Adds auto-completion, error highlighting, integrated debuggers

Note

This course's early examples can be typed and compiled manually to reinforce the compile/run process (Lecture 4-5), then IDEs speed up later work.

Quick Check: Test Your Understanding

- Q: What is the relationship between JDK, JRE, and JVM? (hint: nested)
- Q: Which tool compiles .java files into .class files?
- Q: Why must the JDK's bin/ folder be added to the PATH?

Key Takeaways

- JVM executes bytecode; JRE = JVM + libraries to run programs; JDK = JRE + tools to develop programs
- Key JDK tools: javac (compiler), java (launcher), javadoc (documentation generator), jar (archiver)
- A correctly installed JDK requires JAVA_HOME and PATH to be configured so javac/java work
- Common IDEs for Java development include Eclipse, IntelliJ IDEA, NetBeans, and VS Code

Next Lecture Preview

- Lecture 4 puts this environment to use: writing the structure of a real Java program and tracing the full compilation and execution process step by step.

Questions?

Recap: JDK, JRE, JVM and Environment Setup

- JDK (develop) contains JRE (run) contains JVM (execute bytecode)
- `javac` compiles, `java` launches/executes
- `JAVA_HOME` and `PATH` must be configured for these commands to work from any folder

Course Overview & Today's Agenda

- Anatomy of a basic Java program, line by line
- The special role of the `main()` method
- Class name vs file name: the rule Java enforces
- Tracing compilation (`javac`) and execution (`java`) step by step

Anatomy of a Basic Java Program

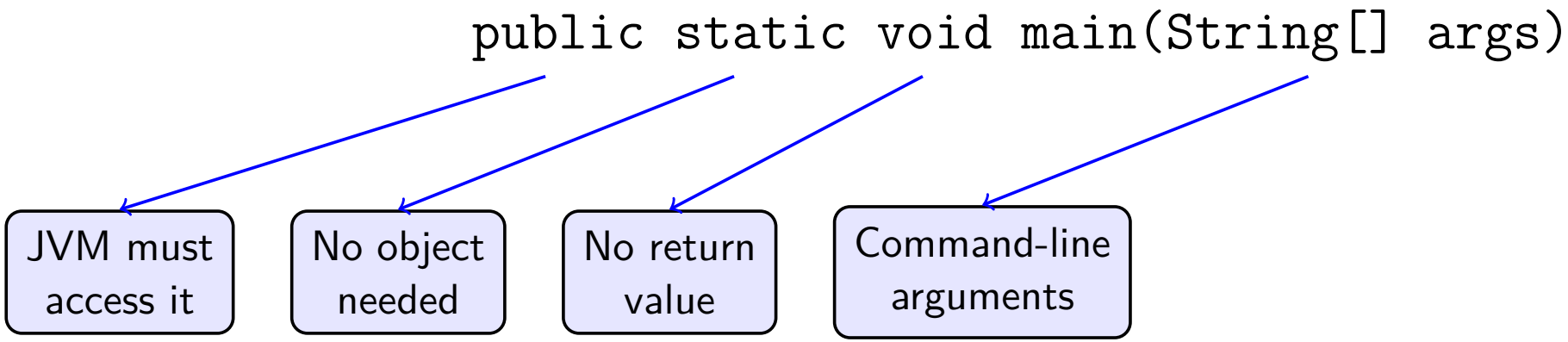
- `public class ClassName { ... }` - every piece of code lives inside a class
- `public static void main(String[] args) { ... }` - the program's entry point
- Statements inside `main()` execute in order, top to bottom
- Curly braces `{ }` define the boundaries of the class and of the method

A Minimal Java Program (Code Walkthrough)

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Why 'public static void main(String[] args)' Exactly?

- **public**: the JVM (outside the class) must be able to call this method, so it cannot be private
- **static**: the JVM calls main() without creating an object of the class first
- **void**: main() does not return any value to the JVM
- **String[] args**: an array holding any command-line arguments passed when the program is run

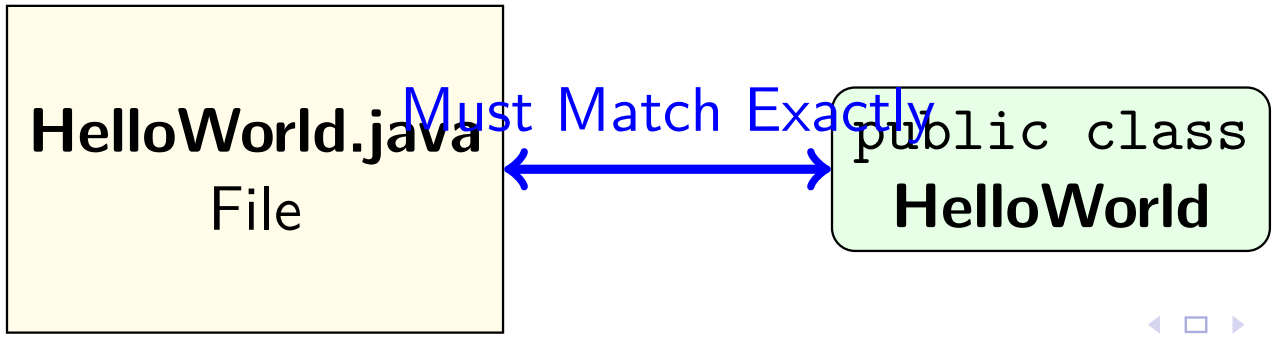


The Class Name vs File Name Rule

Important Rule

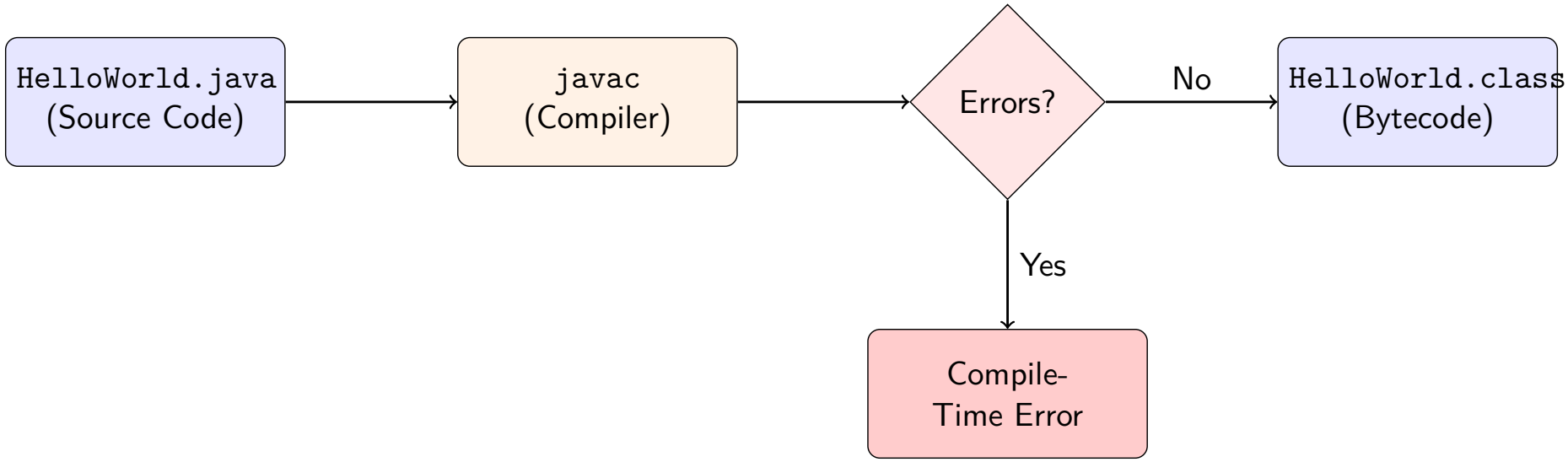
If a class is declared 'public', the file MUST be named exactly ClassName.java (case-sensitive)

- Example: 'public class HelloWorld' must be saved as `HelloWorld.java`
- A single `.java` file may contain multiple classes, but at most one can be public
- Mismatched names cause a compile-time error: *'class HelloWorld is public, should be declared in a file named HelloWorld.java'*



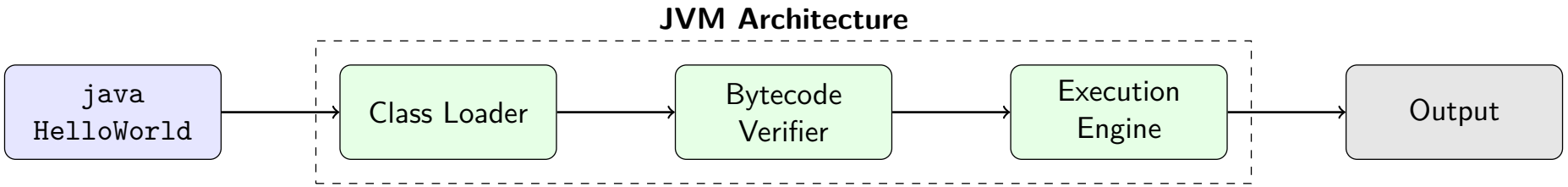
The Compilation Process, Step by Step

- **Step 1:** Save source code as `ClassName.java`
- **Step 2:** Run '`javac ClassName.java`' in the terminal
- **Step 3 & 4:** Checks syntax, produces `ClassName.class` (bytecode) if no errors



The Execution Process, Step by Step

- **Step 1:** Run '`java ClassName`' (no `.class` extension)
- **Step 2 & 3:** JVM loads, verifies, and executes the bytecode
- **Step 4:** Execution begins at `main()`



Compile-Time Errors vs Run-Time Errors

Compile-Time Errors	**Run-Time Errors**
Caught by `javac` before any `.class` file is produced	Occurs while the JVM executes the bytecode
Examples: missing semicolon, misspelled keyword	Examples: dividing by zero, invalid array index
Must be fixed before the program can run at all	Reason Unit 4 introduces exception handling in depth

Common Beginner Compilation Mistakes

- ✗ Forgetting a semicolon at the end of a statement
- ✗ Mismatched curly braces `{ }`
- ✗ File name not matching the public class name exactly (including case)
- ✗ Typing '`java ClassName.class`' instead of just '`java ClassName`' when executing

Quick Check: Test Your Understanding

Questions

1. Why must `main()` be declared '`public static void`'?
2. If a public class is named `Calculator`, what must the file be named?
3. What command compiles a file, and what command runs the compiled program?

Key Takeaways

- Every Java program lives inside at least one class; execution begins at '`public static void main(String[] args)`'
- A public class's file name must exactly match the class name (case-sensitive), e.g., `HelloWorld.java`
- Compilation (`javac`) turns `.java` source into `.class` bytecode; execution (`java`) runs that bytecode on the JVM
- Compile-time errors block bytecode generation entirely; run-time errors occur later, during execution (previewing Unit 4)

Next Lecture Preview

- Lecture 5 completes topic 1.3 with comments and documentation, then ties everything together in a full hands-on Hello World compile-and-run walkthrough.

Questions?

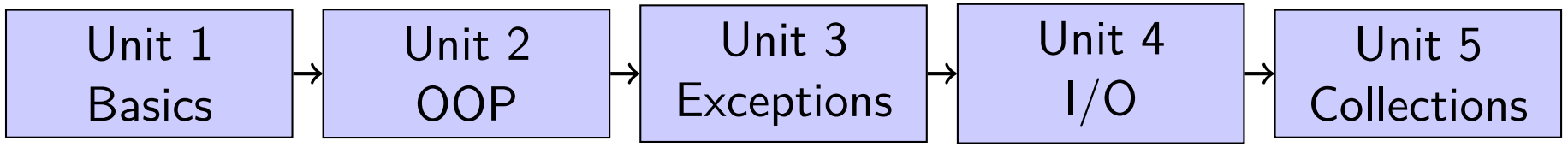
Recap: Program Structure and Compilation

- A Java program lives inside a class; execution starts at public static void main(String[] args)
- File name must exactly match the public class name (e.g., HelloWorld.java)
- javac compiles source to bytecode; java executes that bytecode on the JVM

Course Overview & Today's Agenda

- Types of comments in Java: single-line, multi-line, documentation
- Writing effective Javadoc comments
- Full Hello World walkthrough: type, save, compile, execute
- Topic 1.3 wrap-up: structure + compilation + execution + documentation together

We are here



Single-Line Comments

- Syntax: `// this is a single-line comment`
- Everything after `//` on that line is ignored by the compiler
- Used for brief explanations next to a single statement
- Example: `int total = price * quantity; // compute total cost`

Multi-Line Comments

- Syntax: `/* this comment can span multiple lines */`
- Useful for longer explanations or temporarily disabling a block of code
- Cannot be nested - a `/* ... */` block ends at the first closing `*/` encountered
- Example: `/* This method calculates the area of a rectangle given width and height */`

Documentation Comments (Javadoc)

- Syntax: `/** ... */` placed directly above a class, method, or field
- Processed by the 'javadoc' tool (recall Lecture 3) to auto-generate HTML API documentation
- Special tags: @param (describes a parameter), @return (describes the return value), @author, @since
- Example: `/** Calculates the area of a rectangle. @param w width @param h height @return the area */`

Comments: Quick Comparison

Comment Type	Purpose
`//` (single-line)	Quick inline notes
`/* ... */` (multi-line)	Longer explanations or disabling code block
`/** ... */` (Javadoc)	Formal API documentation processed by jav

Important Note

None of the three comment types affect the compiled .class bytecode

Hello World Walkthrough - Step 1: Write the Code

```java
/** A simple program that prints a greeting. */
public class HelloWorld {
    public static void main(String[] args) {
        // Print a greeting message to the console
        System.out.println("Hello, World!");
    }
}
```

Hello World Walkthrough - Step 2: Save and Compile

- Save the file as exactly HelloWorld.java (matches the public class name, recall Lecture 4)
- Open a terminal in that folder and run: javac HelloWorld.java
- If there are no errors, this silently produces HelloWorld.class in the same folder
- Any typo (missing semicolon, mismatched braces) is reported here as a compile-time error

Hello World Walkthrough - Step 3: Execute

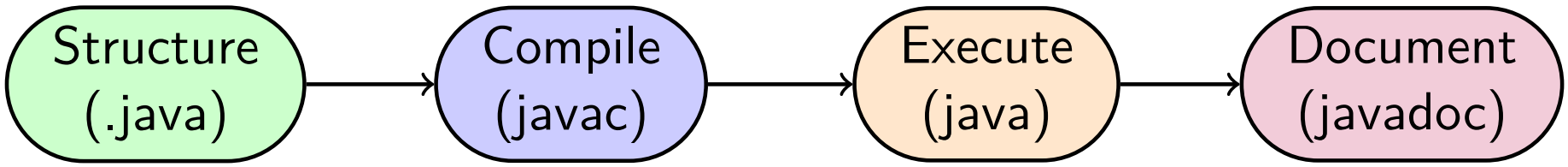
- Run: java HelloWorld (no .class extension, no .java extension)
- The JVM's Class Loader loads HelloWorld.class, verifies it, and executes main()
- Expected output printed to the console: Hello, World!
- If 'java HelloWorld.class' is typed by mistake, the JVM reports an error - always use just the class name

Generating Documentation with javadoc

- Run: javadoc HelloWorld.java
- The javadoc tool scans `/** ... */` comments and generates browsable HTML documentation
- Official Java API docs (docs.oracle.com/javase) are generated using this exact same tool
- Good documentation habits formed now will matter increasingly from Unit 2 onward as classes grow more complex

Topic 1.3 Complete: Full Pipeline Recap

- Structure: class + public static void main(String[] args) (Lecture 4)
- Compilation: javac converts .java source to .class bytecode (Lecture 4)
- Execution: java runs bytecode via the JVM (Lecture 4)
- Documentation: //, /* */, and /** */ comments, with javadoc generating formal docs (Lecture 5)



Quick Check: Test Your Understanding

- Q: Which comment type is processed by the javadoc tool?
- Q: What are the exact terminal commands to compile and then run HelloWorld.java?
- Q: Can /* */ comments be nested? What happens if you try?

Key Takeaways

- Java has three comment types: // (single-line), /* */ (multi-line), and /** */ (Javadoc documentation comments)
- Javadoc comments use tags like @param and @return, and are processed by the javadoc tool into HTML API documentation
- The full Hello World pipeline is: write code -> save as ClassName.java -> javac ClassName.java -> java ClassName
- Comments never affect compiled bytecode - they exist purely to help human readers and documentation tools

Next Lecture Preview

- Lecture 6 begins the final part of Unit 1's topic 1.4: primitive data types and variables in Java.

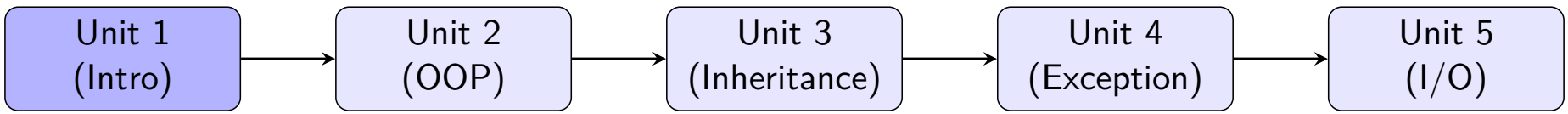
Questions?

Recap: Topic 1.3 Complete

- Java programs compile with javac and execute with java on the JVM
- Comments (//, /* */, /** */) document code without affecting bytecode
- The full pipeline was demonstrated hands-on with Hello World

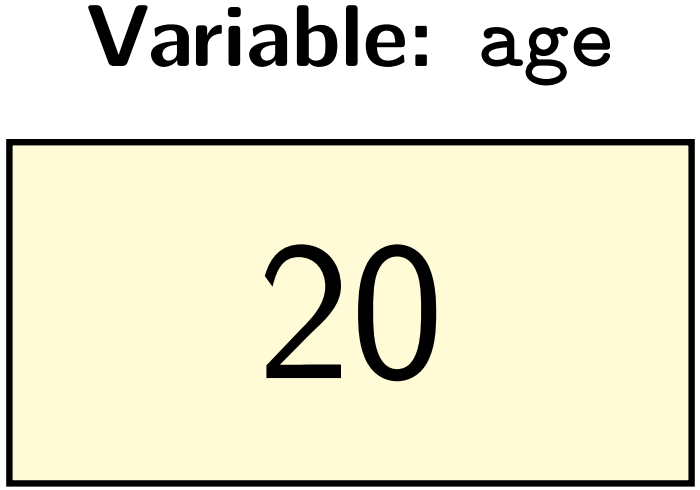
Course Overview & Today's Agenda

- The 8 primitive data types in Java
- Variable declaration and initialization syntax
- Default values and variable naming conventions
- Primitive types vs reference types (a first look, expanded in Unit 2)



What is a Variable?

- A variable is a named storage location in memory that holds a value of a specific type
- Java is a statically-typed language: every variable's type must be declared and cannot change
- General syntax: `dataType variableName = value;`
- Example: `int age = 20;` declares an int variable named age, initialized to 20



The 8 Primitive Data Types: Integer Types

- **byte**: 8-bit, range -128 to 127 - used for memory-critical storage
- **short**: 16-bit, range -32,768 to 32,767 - saves memory in large arrays
- **int**: 32-bit, default integer type (approx -2.1B to 2.1B)
- **long**: 64-bit, for very large integers - suffix 'L'

Type	**Size (bits)**	**Range**
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$

The 8 Primitive Data Types: Floating-Point and Others

- **float**: 32-bit single-precision; requires 'f' suffix (e.g., 3.14f)
- **double**: 64-bit double-precision; default for decimal literals (e.g., 3.14)
- **char**: 16-bit single Unicode character, in single quotes (e.g., 'A')
- **boolean**: holds only true or false

Type	Size (bits)	Example Literal
float	32	3.14159f
double	64	3.1415926535
char	16	'A', '%'
boolean	1 (logical size)	true, false

Declaring and Initializing Variables (Code Example)

```
int age = 20;
double price = 499.99;
char grade = 'A';
boolean isPassed = true;
long population = 8000000000L;
float pi = 3.14159f;
```

Default Values of Primitive Types

Type	Default Value
byte, short, int, long	0
float, double	0.0
char	'\u0000' (null character)
boolean	false

Local Variables Have No Default Value

- A local variable (declared inside a method, like main()) is NOT automatically initialized
- Using a local variable before assigning it a value causes a compile-time error
- This is a deliberate Java safety feature to prevent accidental use of garbage values

```
public static void main(String[] args) {
    int x;
    // Error: variable x might not have been initialized
    System.out.println(x);
}
```

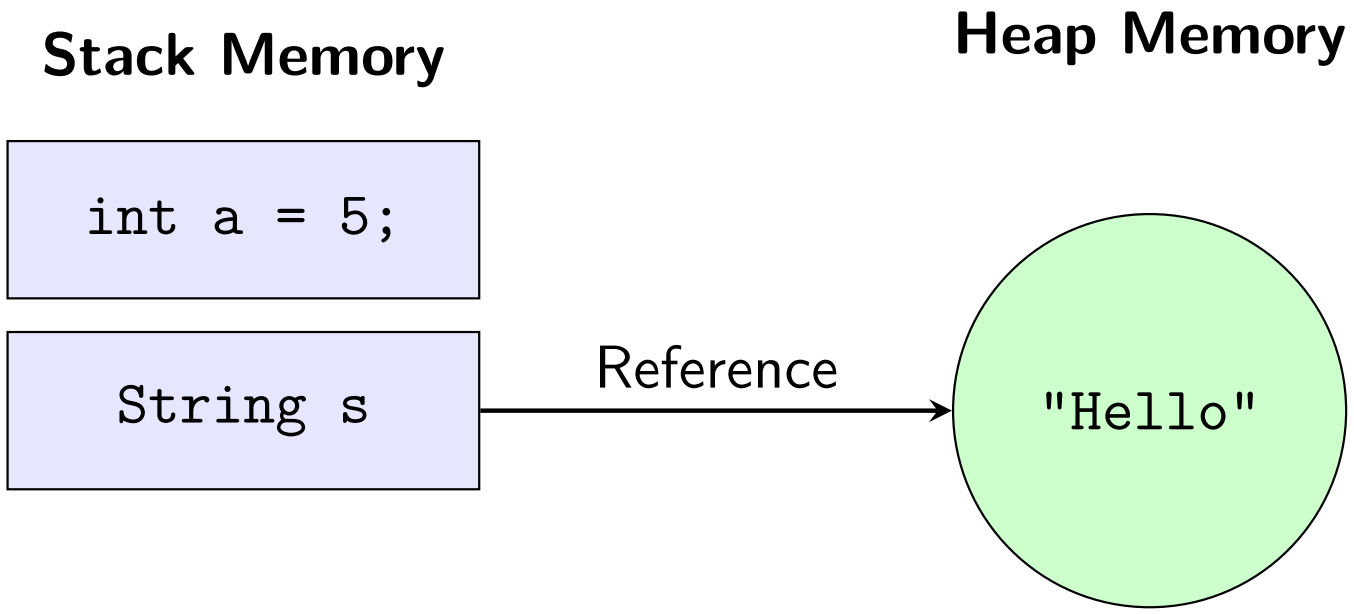
Variable Naming Conventions

- Must start with a letter, underscore (_), or dollar sign ($)
- Convention: camelCase for variables and methods (e.g., totalAmount)
- Cannot use Java reserved keywords (e.g., class, int, public)
- Names should be meaningful: `firstName` is better than `fn`

Valid Names	**Invalid Names**
`age`	`1stPlace` (starts with digit)
`_totalValue`	`my-name` (hyphen not allowed)
`$price`	`class` (reserved keyword)
`studentName`	`student name` (contains space)

Primitive Types vs Reference Types: A First Look

- **Primitive types** store the actual value directly in memory (stack).
- **Reference types** store a reference (address) to an object on the heap.
- For now: primitives are simple values; everything else is accessed via reference.



Quick Check: Test Your Understanding

- Q: Name the 8 primitive data types in Java.
- Q: What literal suffix is required for a long value? For a float value?
- Q: Does a local variable get a default value automatically? What about an instance field?

Key Takeaways

- Java has 8 primitive types: byte, short, int, long, float, double, char, boolean
- int is the default type for whole-number literals; double is the default type for decimal literals
- Instance fields get automatic default values; local variables do NOT and must be explicitly initialized
- Primitive types store values directly; reference types store a reference to an object on the heap

Next Lecture Preview

- Lecture 7 covers type conversion and casting: how values move between different primitive types, both automatically and explicitly.

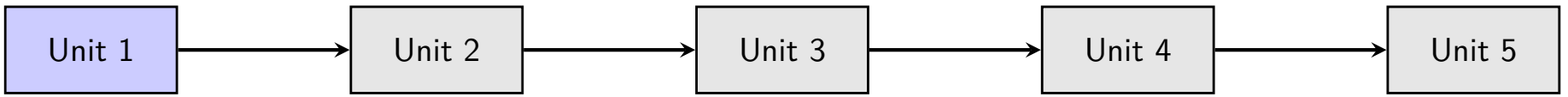
Questions?

Recap: Primitive Data Types

- 8 primitive types: byte, short, int, long, float, double, char, boolean
- int is the default integer type; double is the default decimal type
- Local variables have no default value and must be explicitly initialized

Course Overview & Today's Agenda

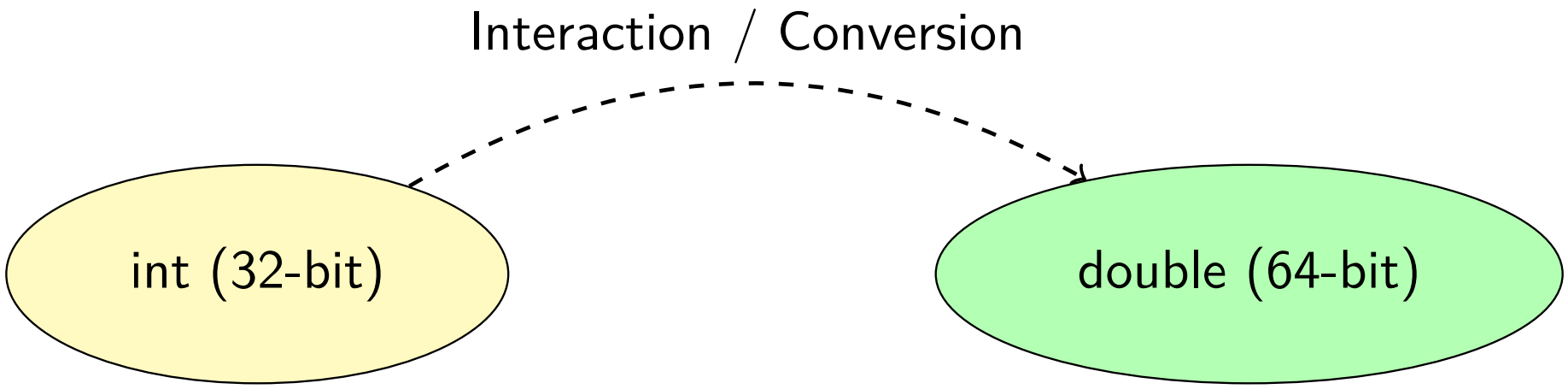
- Implicit (widening) type conversion
- Explicit (narrowing) type conversion / casting
- Data loss and overflow scenarios during casting
- Automatic type promotion inside expressions



We are here

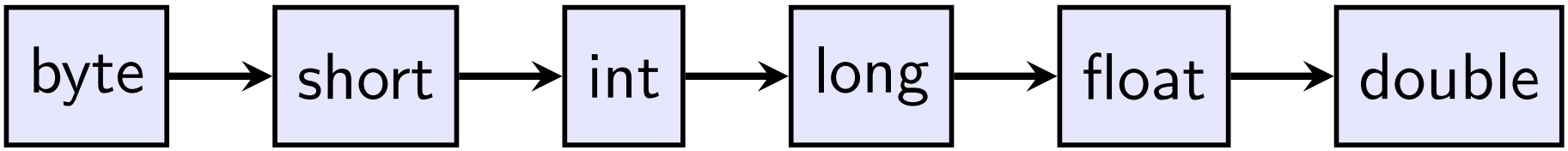
Why Type Conversion is Needed

- Java is statically typed - every variable has a fixed type, but programs often need to combine values of different types
- Example: adding an int and a double together, or storing a large 'long' result into a smaller variable
- Java defines strict rules for when conversion happens automatically and when the programmer must do it explicitly
- This matters directly for the operators covered in Lecture 8



Implicit (Widening) Type Conversion

- Happens automatically when converting a SMALLER type to a LARGER, compatible type
- Order (smallest to largest): byte $\rightarrow$ short $\rightarrow$ int $\rightarrow$ long $\rightarrow$ float $\rightarrow$ double
- No data is lost because the destination type can always represent the source type's full range
- Example: `int wholeMarks = 85; double marks = wholeMarks;`



Widening Conversion: Code Example

```
byte b = 10;
int i = b;        // byte to int: automatic widening
long l = i;       // int to long: automatic widening
float f = l;      // long to float: automatic widening
double d = f;     // float to double: automatic widening
System.out.println(d); // prints 10.0
```

Explicit (Narrowing) Type Conversion / Casting

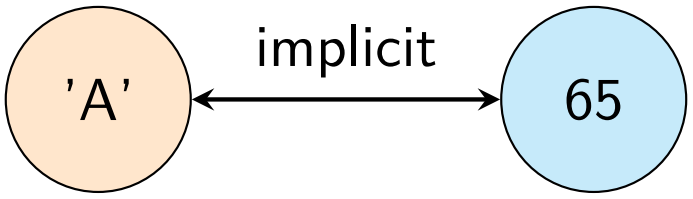
- Required when converting a LARGER type to a SMALLER type, since data could be lost
- Syntax: `destinationType variable = (destinationType) sourceValue;`
- The programmer must explicitly acknowledge the possible data loss using the cast operator
- Example: `double price = 99.99; int roundedDown = (int) price;`

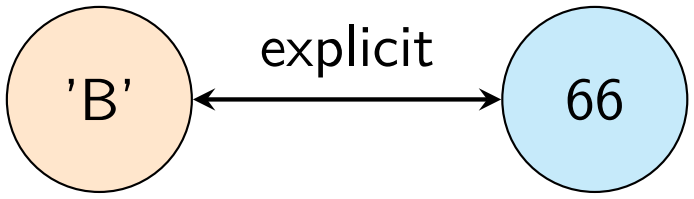
Narrowing Conversion: Data Loss Examples

```
double d = 9.99;
int i = (int) d;          // i = 9 (decimal part truncated, NOT
    rounded)
int big = 130;
byte small = (byte) big; // small = -126 (overflow wraps around,
     since byte max is 127)
```

Casting Between char and int

- `char` is internally stored as a 16-bit Unicode numeric code, so it can convert to/from `int`
- `char c = 'A'; int code = c;` (implicit widening, code becomes 65)
- `int code2 = 66; char c2 = (char) code2;` (explicit narrowing, c2 becomes 'B')
- This char↔int relationship is useful for simple character arithmetic





Automatic Type Promotion in Expressions

- byte, short, and char operands are automatically promoted to int during arithmetic expressions
- If either operand in an expression is long/float/double, the whole expression promotes to that wider type

Type Promotion Example

```
byte a = 10;
byte b = 20;
// byte c = a + b; // COMPILE ERROR: a+b is promoted to
    int
byte c = (byte)(a + b); // explicit cast required
```

Widening vs Narrowing: Quick Comparison

Widening	**Narrowing**
Small type → Large type	Large type → Small type
Automatic, no cast syntax needed	Must be explicit (`targetType`)
No data loss	Possible data loss (truncation/overflow)
`byte` → `short` → `int` → `long` → `float` → `double`	Simply the reverse direction

Quick Check: Test Your Understanding

- **Q:** Is int-to-double conversion implicit or explicit? What about double-to-int?
- **Q:** What is the result of `(int) 9.99`? Is it rounded or truncated?
- **Q:** Why does `byte c = a + b;` fail to compile even if a and b are both byte?

Key Takeaways

- **Implicit (widening):** Happens automatically (byte $\rightarrow$ short $\rightarrow$ int $\rightarrow$ long $\rightarrow$ float $\rightarrow$ double) with no data loss
- **Explicit (narrowing):** Requires cast syntax `(targetType) value` and can lose data via truncation or overflow
- **char and int:** Can convert between each other because `char` is stored as a 16-bit Unicode numeric code
- **Type Promotion:** byte/short/char operands automatically promote to `int` in expressions, often requiring an explicit cast to store back

Next Lecture Preview

- Lecture 8 covers Java's arithmetic, relational, logical, and bitwise operators, building directly on today's type promotion rules.

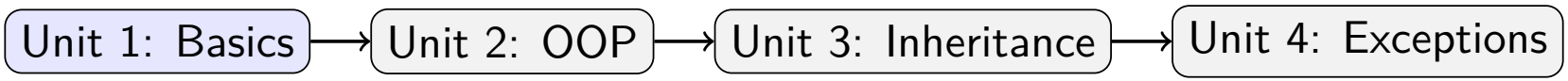
Questions?

Recap: Type Conversion and Casting

- Widening conversion (small to large type) is automatic; narrowing requires an explicit (type) cast
- Narrowing can lose data via truncation (decimals dropped) or overflow (value wraps around)
- byte/short/char operands are automatically promoted to int inside expressions

Course Overview & Today's Agenda

- Arithmetic operators and integer vs floating-point division
- Relational operators
- Logical operators and short-circuit evaluation
- Bitwise operators



Arithmetic Operators

- + addition, - subtraction, * multiplication, / division, % modulus (remainder)

```
int a = 7, b = 2;
int res1 = a / b; // gives 3 (integer division discards
    remainder)
int res2 = a % b; // gives 1 (the remainder of 7 divided by 2)

double x = 7.0, y = 2.0;
double res3 = x / y; // gives 3.5 (floating-point division)
```

Increment and Decrement Operators

```
int i = 5;
i++;            // post-increment: i becomes 6
++i;            // pre-increment: i becomes 7

// Context matters in assignment:
int j = i++; // j gets 7 (old value), then i becomes 8
int k = ++i; // i becomes 9 first, then k gets 9
```

Relational Operators

- Every relational operator produces a boolean result: `true` or `false`
- Relational operators are the foundation of conditional statements (if/else) used throughout every program

Operator	Meaning	Example Expression → Result
`==`	equal to	`5 == 5` → `true`
`!=`	not equal to	`5 != 5` → `false`
`>`	greater than	`7 > 2` → `true`
`<`	less than	`7 < 2` → `false`
`>=`	greater or equal	`4 >= 4` → `true`
`<=`	less or equal	`3 <= 2` → `false`

Logical Operators

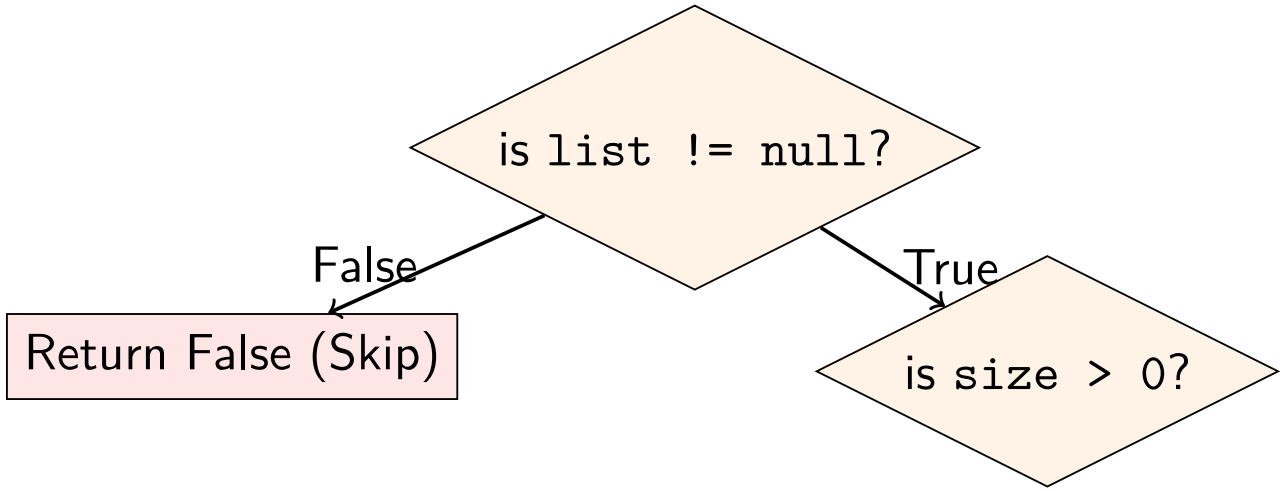
- `&&` logical AND: `true` only if BOTH operands are `true`
- `||` logical OR: `true` if AT LEAST ONE operand is `true`
- `!` logical NOT: reverses a boolean value (`true` becomes `false` and vice versa)
- Example: `boolean eligible = (age >= 18) && (hasID == true);`

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A || B
T	T	T
T	F	T
F	T	T
F	F	F

Short-Circuit Evaluation

- `&&` and `||` use short-circuit evaluation: the second operand is skipped if the result is already determined
- For `&&`: if the first operand is `false`, the whole expression is `false` - second operand is never evaluated
- For `||`: if the first operand is `true`, the whole expression is `true` - second operand is never evaluated
- Example: `(list != null) && (list.size() > 0)` safely avoids a NullPointerException



Bitwise Operators

- & bitwise AND, | bitwise OR, ^ bitwise XOR, ~ bitwise complement (NOT)
- << left shift (multiplies by 2^n), >> right shift (divides by 2^n)
- These operate on the individual binary bits of integer types

```
a = 5 = 0 1 0 1
b = 3 = 0 0 1 1
-----------------
& = 1 = 0 0 0 1
| = 7 = 0 1 1 1
^ = 6 = 0 1 1 0
```

Bitwise Operators: A Practical Example

```c
int flags = 0;

// set bit 0 (turn a feature ON)
flags = flags | 1;

// clear bit 0 (turn that feature OFF)
flags = flags & ~1;

// left shift by 1 = 8 (equivalent to 4 * 2)
int doubled = 4 << 1;
```

Operator Categories: Quick Comparison

Category	Operators	Result Type
Arithmetic	`+, -, *, /, %`	Numeric
Relational	`==, !=, >, <, >=, <=`	Boolean
Logical	`&&, ||, !`	Boolean
Bitwise	`&, |, ^, ~, <<, >>`	Integer (bits)

Quick Check: Test Your Understanding

- **Q:** What is the result of `7 / 2` in integer division? What about `7.0 / 2.0`?
- **Q:** Why does `(a != null) && (a.value > 0)` avoid a crash when `a` is null?
- **Q:** Which operator category produces boolean results: relational or bitwise?

Key Takeaways

- Arithmetic operators include `+ - * / %`; integer division truncates decimals, floating-point division keeps them
- Relational operators (`==, !=, >, <, >=, <=`) always produce a boolean result
- Logical operators (`&&, ||, !`) combine booleans and use short-circuit evaluation to skip unnecessary evaluation
- Bitwise operators (`&, |, ^, ~, <<, >>`) manipulate the individual binary bits of integer values

Next Lecture Preview

- Lecture 9 covers operator precedence and associativity - the rules that decide evaluation order when multiple operators appear in one expression - and wraps up Unit 1.

Questions?

Recap: The Four Operator Categories

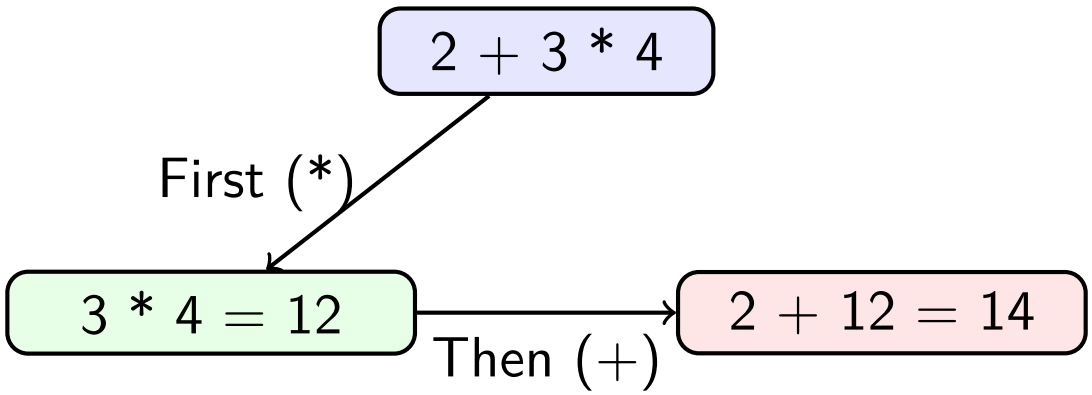
- Arithmetic: + - * / % (numeric results)
- Relational: == != > < >= <= (boolean results)
- Logical: && || ! (boolean combination, short-circuit evaluation)
- Bitwise: & | ^ ~ « » (bit-level manipulation)

Course Overview & Today's Agenda

- Operator precedence: which operators evaluate first
- Associativity: left-to-right vs right-to-left evaluation
- Using parentheses to remove ambiguity
- Unit 1 wrap-up: Java fundamentals recap before Unit 2's OOP concepts begin

What is Operator Precedence?

- When an expression has multiple operators, precedence decides which one is evaluated first
- Higher-precedence operators bind more tightly than lower-precedence ones
- Example: $2 + 3 * 4$ evaluates as $2 + (3 * 4) = 14$, NOT $(2 + 3) * 4 = 20$, because * has higher precedence than +
- This mirrors standard mathematical order of operations (similar to PEMDAS/BODMAS)



Java's Precedence Order (High to Low, Simplified)

Level	**Operators**
Highest	Postfix (++, –), Unary (+, -, !, ~), (type)cast
High	* / %
Medium	+ -
Medium-low	Relational (< > <= >=), Equality (== !=)
Low	Logical (&& then ||)
Lowest	Assignment (= += -= etc.)

Precedence Example: Mixing Arithmetic and Relational

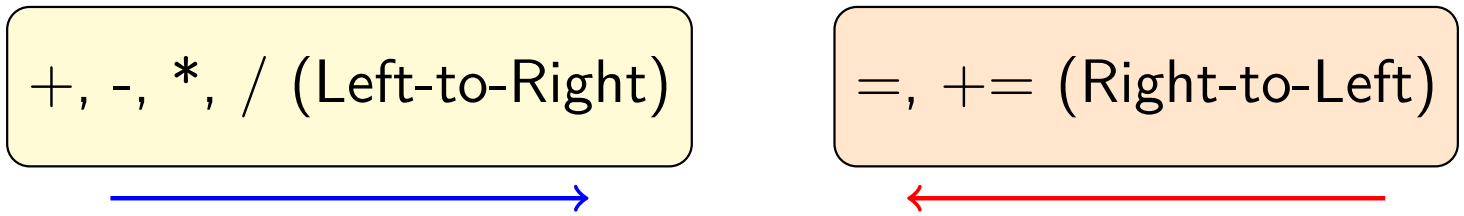
```
int a = 5, b = 10, c = 2;
boolean result = a + b > c * 5;
// Step 1: c * 5 = 10 (multiplication first)
// Step 2: a + b = 15 (addition next)
// Step 3: 15 > 10 evaluates to true (relational last)
```

Precedence Example: Mixing Logical Operators

```
boolean x = true, y = false, z = true;
boolean result = x || y && z;
// && has higher precedence than ||, so this means: x || (y && z
    )
// Step 1: y && z = false && true = false
// Step 2: x || false = true || false = true
```

What is Associativity?

- Associativity decides evaluation order when operators of the SAME precedence appear together
- Most binary operators (+, -, *, /, relational, logical) are left-to-right associative
- Assignment operators (=, +=, -=, etc.) are right-to-left associative
- Example: a = b = c; assigns c to b first, then assigns that result to a (right-to-left)



Associativity Example: Left-to-Right

```
int result = 20 - 5 - 3;
// Both '-' operators have equal precedence, so left-to-right
    associativity applies:
// Step 1: 20 - 5 = 15
// Step 2: 15 - 3 = 12
// result = 12 (NOT 20 - (5 - 3) = 18)
```

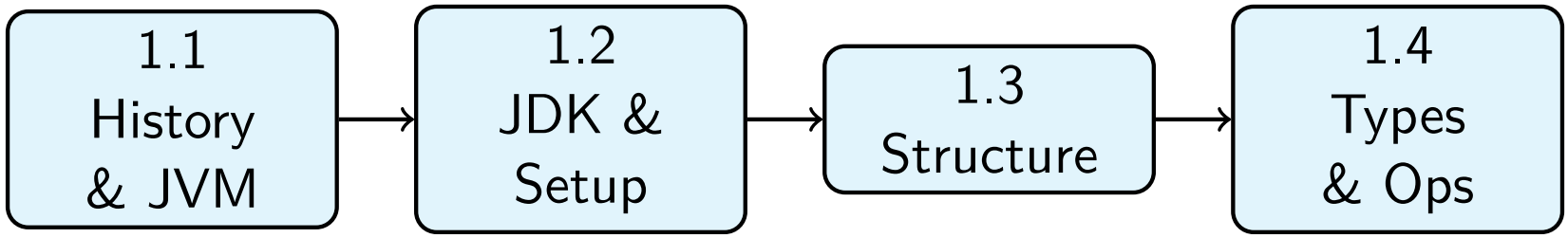
Using Parentheses to Remove Ambiguity

- Parentheses ( ) always evaluate first and can override default precedence
- Even when default precedence gives the 'correct' order, parentheses make code more readable
- Best practice: use parentheses generously in complex expressions, even when technically optional

```
// Unclear (relies heavily on precedence rules)
boolean result = a + b > c * 5;

// Clearer (explicitly groups operations)
boolean result = (a + b) > (c * 5);
```

Unit 1 Wrap-Up: The Complete Picture

- 1.1: Java's history, features, and JVM architecture (write once, run anywhere)
- 1.2: JDK/JRE/JVM relationship and environment setup (javac, java, IDEs)
- 1.3: Program structure, compilation/execution process, and comments/documentation
- 1.4: Primitive data types, operators, and today's precedence/associativity rules



Quick Check: Test Your Understanding

Questions

- Q: In 2 + 3 * 4, which operator evaluates first, and why?
- Q: Is assignment (=) left-to-right or right-to-left associative?
- Q: Why is it good practice to use parentheses even when precedence already gives the intended result?

Key Takeaways

- Operator precedence decides which operator evaluates first when multiple operators appear together (e.g., * before +)
- Associativity decides order among equal-precedence operators: most operators are left-to-right, but assignment is right-to-left
- Parentheses always evaluate first and should be used generously to make code intent unambiguous
- Unit 1 built Java's foundation - all of which Unit 2's OOP concepts now build upon

Next Lecture Preview

- Lecture 10 begins Unit 2 (Object-Oriented Programming Concepts)
- Fundamentals of OOP: classes, objects, encapsulation, inheritance, and polymorphism
- Contrasted with procedural programming

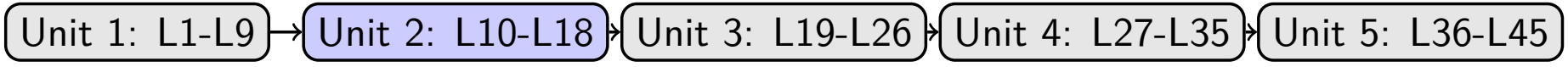
Questions?

Recap: Lecture 9 and Unit 1 Wrap-Up

- Operator precedence and associativity govern how complex Java expressions are evaluated
- Unit 1 wrap-up: Java history/features/JVM (1.1), JDK/JRE/setup (1.2), program structure/compilation/execution (1.3), data types/casting/operators (1.4)

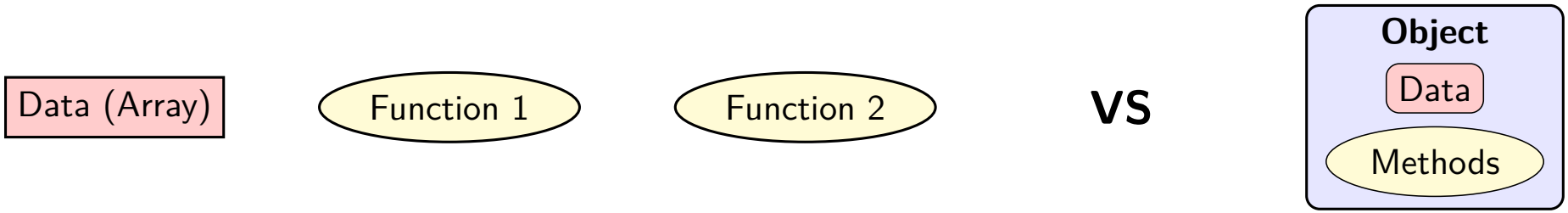
Course Overview & Today's Agenda

- Unit 2 roadmap: Lectures 10-18 on Object-Oriented Programming Concepts
- Why object-oriented programming? Procedural vs OOP
- The five core OOP terms: class, object, encapsulation, inheritance, polymorphism
- Where Unit 2 is headed: classes/objects (11-12), access modifiers (13-14), methods/constructors (15-16), overloading/static/final (17-18)



Why Move Beyond Procedural Programming?

- Procedural programs are organized as a sequence of functions that operate on separate data
- As programs grow, it becomes hard to track which function touches which data, and data can be changed from anywhere
- Object-oriented programming (OOP) instead groups data and the functions that operate on it into a single unit called an object
- This mirrors how we naturally think about real-world entities: a 'Student' has both data (name, roll number) and behavior (calculateGrade())



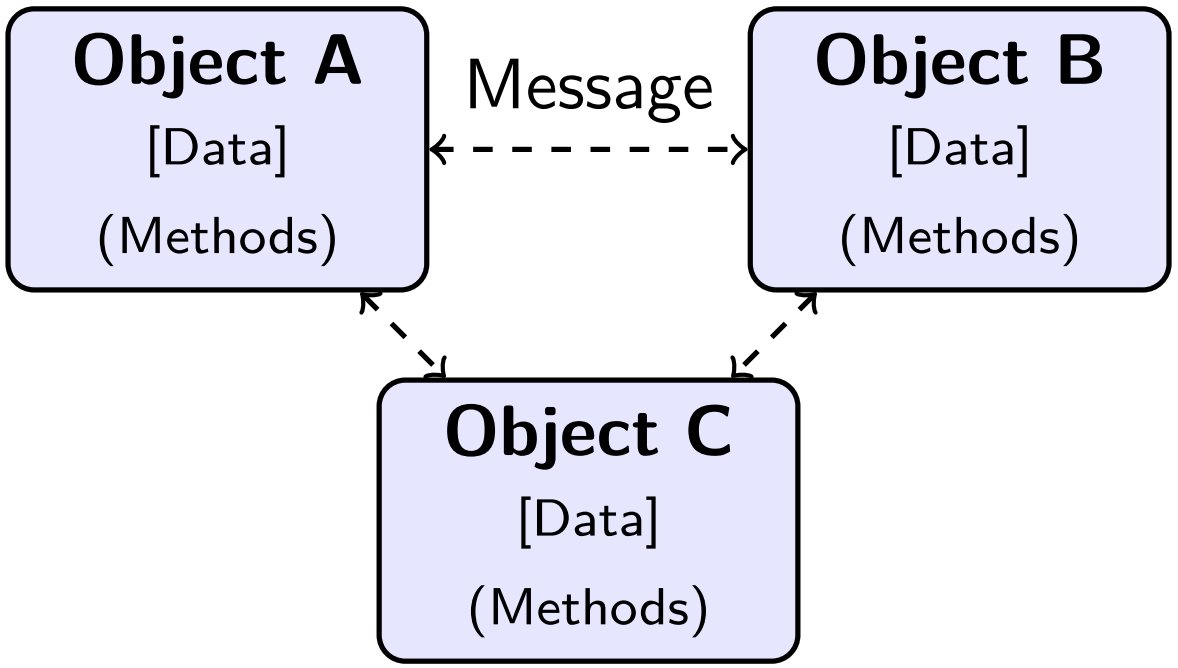
Procedural Programming: A Quick Look

- Procedural style: data and logic are separate

```
// Procedural style: data and logic are separate
String[] names = {"Asha", "Ravi"};
int[] marks = {88, 74};

static void printReport(String[] n, int[] m, int i) {
    System.out.println(n[i] + ": " + m[i]);
}
```

Object-Oriented Programming: The Core Idea

- OOP organizes a program as objects combining state and behavior
- Every object is an instance of a class, which acts as its blueprint
- Objects communicate by calling each other's methods
- Java is object-oriented - almost everything lives inside a class

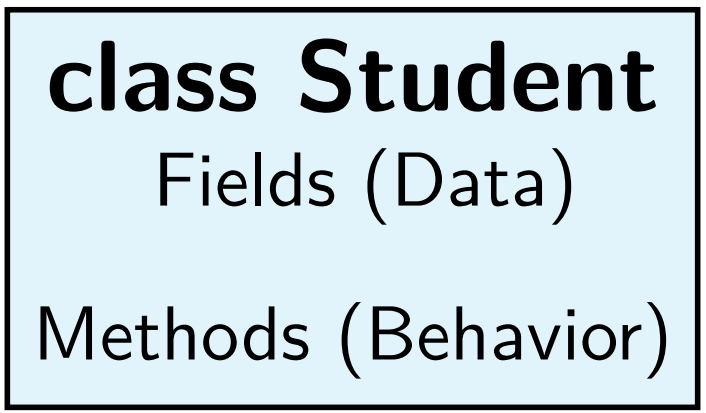


Procedural vs. Object-Oriented Programming

Procedural Programming	**Object-Oriented Programming (OOP)**
Program is a set of functions; data is often global or passed around explicitly	Program is a set of interacting objects; data is bundled inside objects and accessed through methods
Adding a new data type often means editing many existing functions	Adding a new type of object usually means adding a new class, leaving existing classes untouched

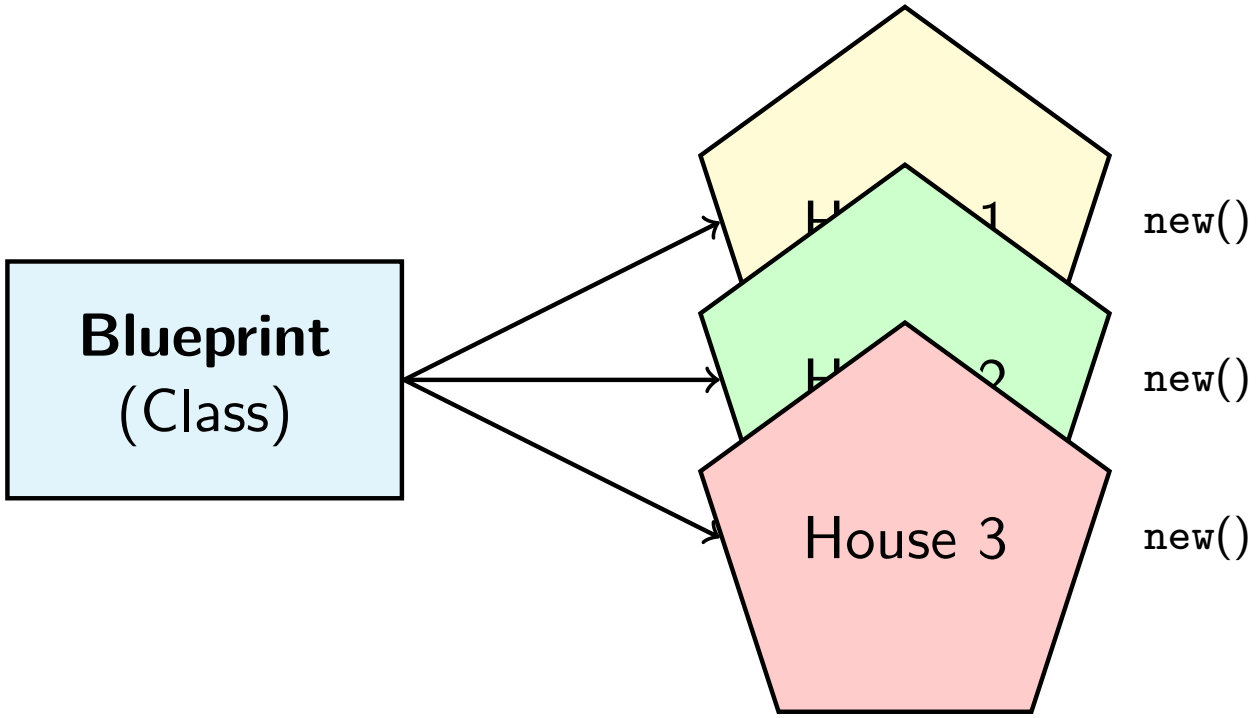
Class: The Blueprint

- A class is a user-defined blueprint that describes what data (fields) and behavior (methods) its objects will have
- A class itself does not occupy object memory - it is a template, like an architectural blueprint for a house
- Java syntax: `class Student { /* fields and methods go here */ }`
- Full class syntax and a working example are covered in detail in Lecture 11



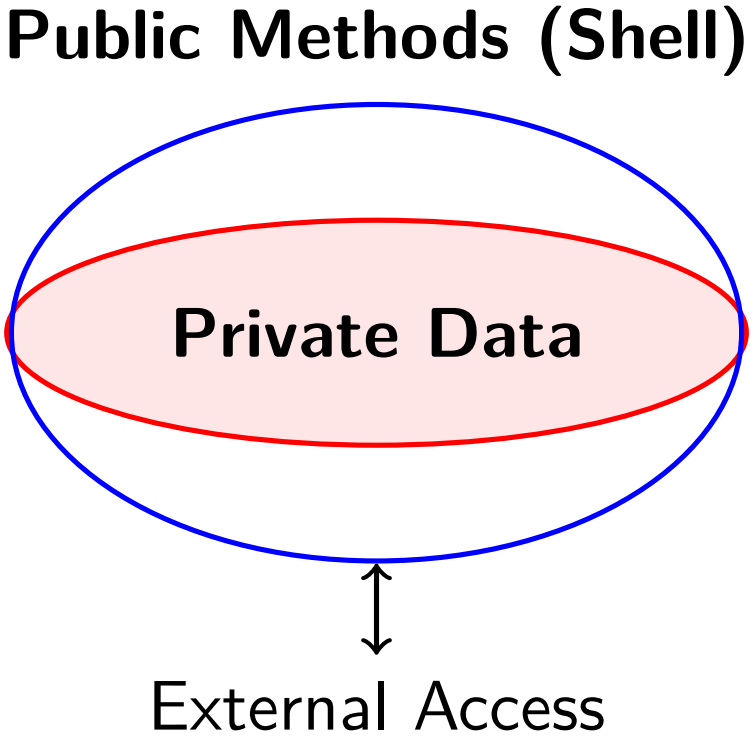
Object: An Instance of a Class

- Object: a concrete instance created in memory using `new`
- Multiple objects can be created, each with independent fields
- Analogy: if Student is the blueprint, `s1 = new Student()` is one actual house
- Covered in full detail in Lectures 11-12



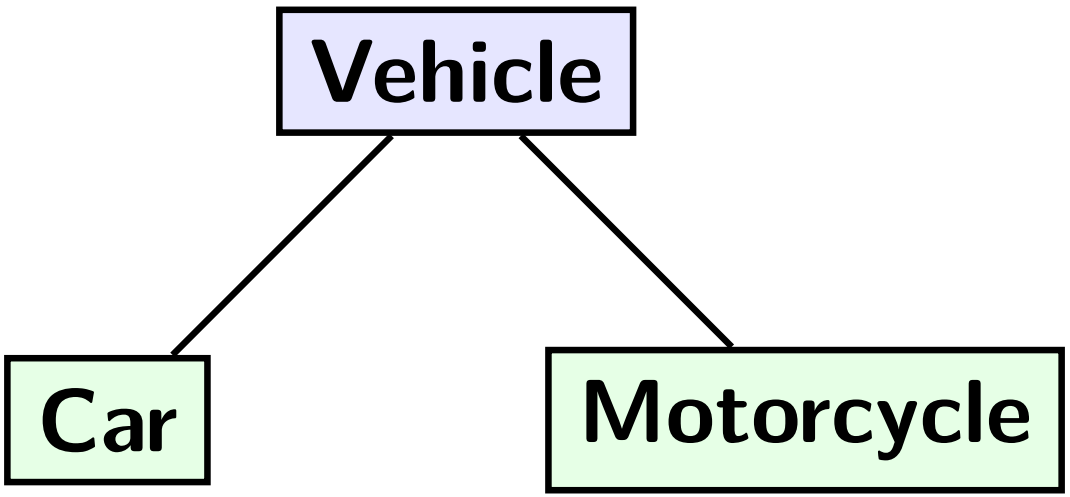
Encapsulation: Bundling and Protecting Data

- Encapsulation: bundling data with methods and restricting direct access
- Achieved using access modifiers and getter/setter methods
- Benefit: internal data stays consistent via controlled methods
- Covered in detail in Lectures 13-14



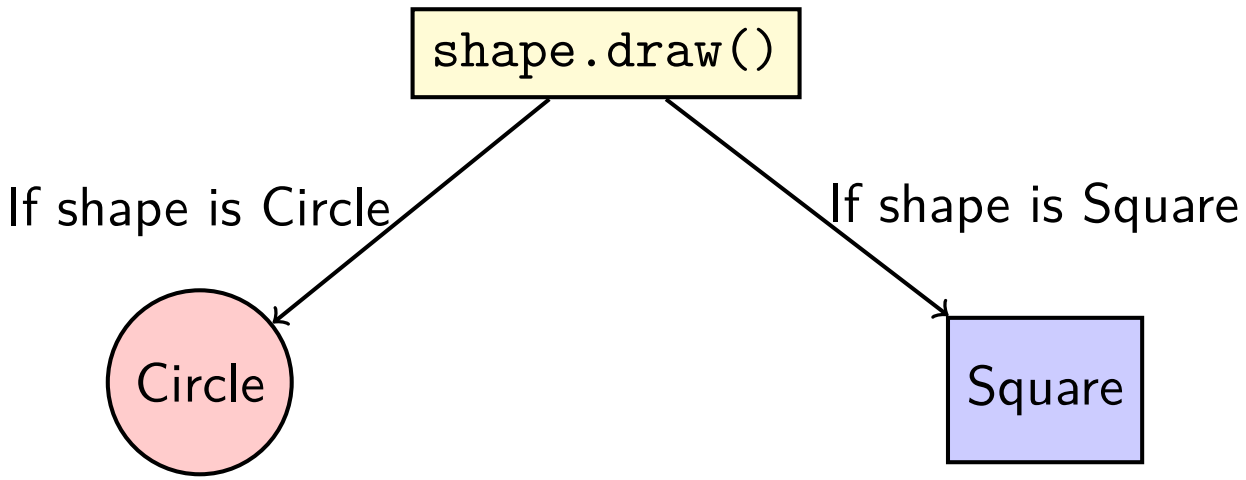
Inheritance: Reusing and Extending

- Inheritance lets one class (subclass) acquire the fields and methods of another class (superclass)
- Promotes code reuse: common behavior is written once in a general class and reused by more specific classes
- Example: a general Vehicle class could be extended by Car and Motorcycle classes
- This is the entire subject of Unit 3, starting at Lecture 19 with the `extends` keyword



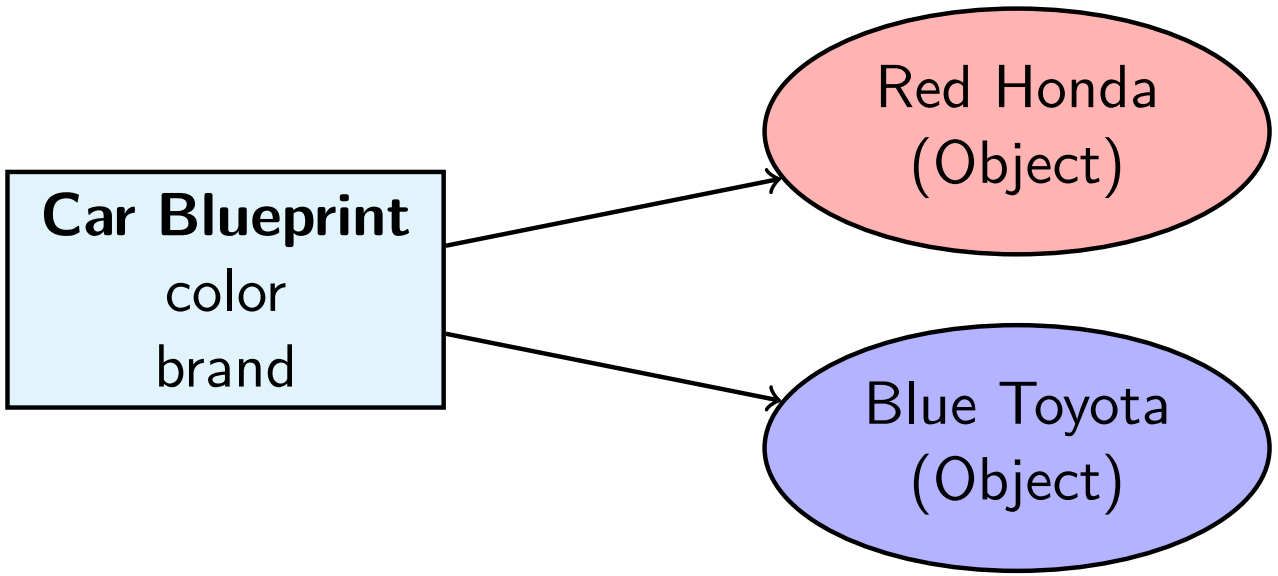
Polymorphism: One Interface, Many Forms

- Polymorphism means the same method name or interface can behave differently depending on the object or arguments involved
- Java supports two kinds: compile-time (method overloading, Lecture 17) and runtime (method overriding, Unit 3 Lecture 20)
- Example: a single draw() method could behave differently for a Circle object versus a Square object
- Poly = many, morph = forms - literally 'many forms' of the same operation



Real-World Analogy: Modeling a Car as a Class

- class Car (blueprint) defines: fields - color, brand, speed; methods - accelerate(), brake(), honk()
- myCar = new Car() creates one specific red Honda; yourCar = new Car() creates a separate blue Toyota
- Each Car object has its own color and speed, but shares the same accelerate()/brake()/honk() behavior defined by the class
- This single analogy will be turned into real, compilable Java code in Lecture 11 using a Student class



The Five Pillars at a Glance

Pillar	Description
Class	Blueprint/template describing fields and methods (no memory used by itself)
Object	A runtime instance of a class, created with `new`, occupying its own memory
Encapsulation	Bundling data with methods and restricting direct access (Lectures 13-14)
Inheritance	Acquiring fields/methods of another class via `extends` (Unit 3, Lecture 19+)
Polymorphism	One method name behaving differently: overloading (Lecture 17) or overriding (Unit 3)

Key Takeaways

- OOP models real-world entities as objects that bundle data (state) and behavior (methods), unlike procedural programming's separate functions and data
- The five foundational OOP terms are class, object, encapsulation, inheritance, and polymorphism
- A class is a blueprint; an object is a concrete instance of that blueprint created at runtime with `new`
- Java is a purely object-oriented language, so every program from here onward will be built using these five concepts

Next Lecture Preview

- Lecture 11 puts these ideas into practice: defining an actual Java class with fields and methods, and creating objects from it using the `new` keyword.

Questions?

Recap: OOP Fundamentals

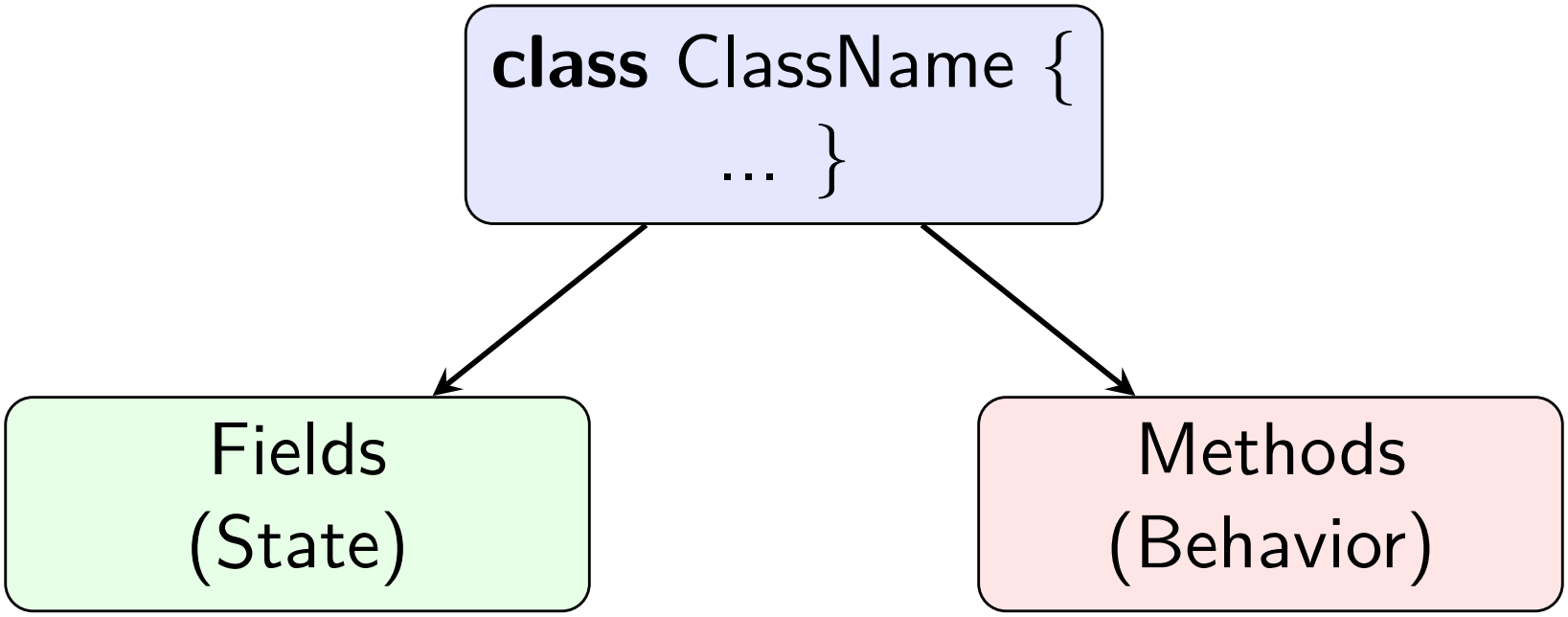
- OOP bundles data (fields) and behavior (methods) together inside objects
- The five pillars: class, object, encapsulation, inheritance, polymorphism
- OOP differs from procedural programming by organizing code around interacting objects rather than free-standing functions

Course Overview & Today's Agenda

- Anatomy of a Java class: fields and methods
- Declaring a class and its instance variables
- Creating objects with the `new` keyword
- Accessing fields and methods via the dot operator

Anatomy of a Java Class

- A class definition begins with the `class` keyword followed by the class name (capitalized by convention)
- Inside the class body: instance variables (fields) declare the object's state
- Also inside the class body: methods declare the object's behavior



Defining the Student Class

```java
public class Student {
    // instance variables (fields)
    String name;
    int rollNo;
    int marks;

    // method (behavior)
    void displayInfo() {
        System.out.println(rollNo + " " + name + " " + marks);
    }
}
```

Instance Variables: Each Object's Own Copy

- Fields declared inside a class are called instance variables
- Every object gets its own independent copy of these variables

Student Object 1
name = "Asha"
rollNo = 21
marks = 88

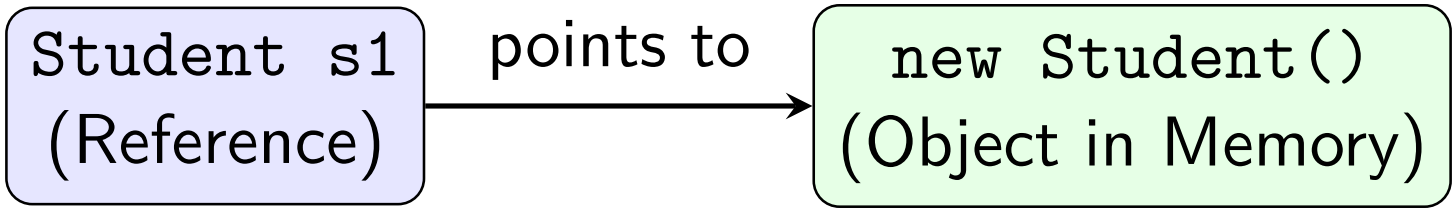
Student Object 2
name = "Ravi"
rollNo = 15
marks = 74

Default Values

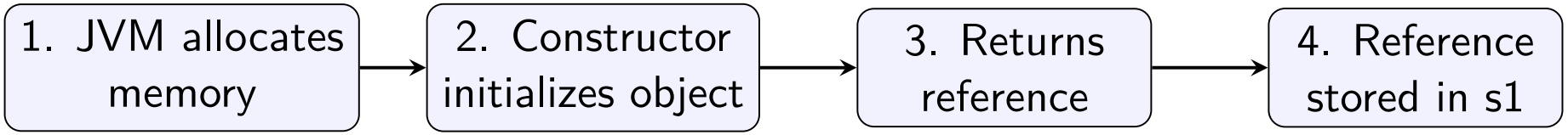
Unlike local variables, instance variables get automatic default values (0, null, false) if not initialized.

Creating an Object: the `new` Keyword

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
    }
}
```



What Happens When `new` Runs



- Memory is allocated with default values
- The constructor runs
- `new` returns the address which is stored in the reference variable

Accessing Fields and Methods: the Dot Operator

```
Student s1 = new Student();
s1.name = "Asha";
s1.rollNo = 21;
s1.marks = 88;
s1.displayInfo();  // prints: 21 Asha 88
```

Dot Operator

Use `objectReference.memberName` to access fields or methods of an object.

Multiple Objects, Independent State

```
Student s1 = new Student();
Student s2 = new Student();

s1.name = "Asha"; s1.marks = 88;
s2.name = "Ravi"; s2.marks = 74;

s1.displayInfo();  // 0 Asha 88
s2.displayInfo();  // 0 Ravi 74
```

s1
name = "Asha"
rollNo = 0
marks = 88

s2
name = "Ravi"
rollNo = 0
marks = 74

Class vs. Object: Terminology Check

Class	**Object**
Declared once with `class` keyword	Created many times with `new`
A compile-time blueprint	A runtime instance
No memory allocated for fields	Occupies its own memory for fields
Example name: `Student` (capitalized)	Example name: `s1`, `s2` (lowercase)

Common Mistakes with Classes and Objects

- **Forgetting `new`**: `Student s1;` only declares a reference, it does NOT create an object (`s1` is null)
- Calling a method or accessing a field on a `null` reference causes a `NullPointerException` at runtime
- Confusing the class name with the object/reference variable name in code
- Forgetting that uninitialized instance variables get default values (0 / null / false), not garbage values

Putting It Together: A Complete Program

```
public class Student {
    String name; int rollNo; int marks;
    void displayInfo() {
        System.out.println(rollNo + " " + name + " " + marks);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Asha"; s1.rollNo = 21; s1.marks = 88;
        s1.displayInfo();
    }
}
```

Key Takeaways

- A Java class groups related fields (state) and methods (behavior) under one blueprint name
- Objects are created from a class using the `new` keyword, which allocates memory and returns a reference
- Each object maintains its own independent copy of instance variables
- The dot (.) operator accesses an object's fields and methods through its reference variable

Next Lecture Preview

- Lecture 12 goes under the hood of what `new` actually does - where objects live in memory (heap vs stack) and how Java's garbage collector reclaims objects no longer referenced.

Questions?

Recap: Classes and Objects

- A class defines fields and methods; an object is created from it using `new`
- The dot operator accesses an object's members: s1.name, s1.displayInfo()
- Each object holds an independent copy of instance variables

Course Overview & Today's Agenda

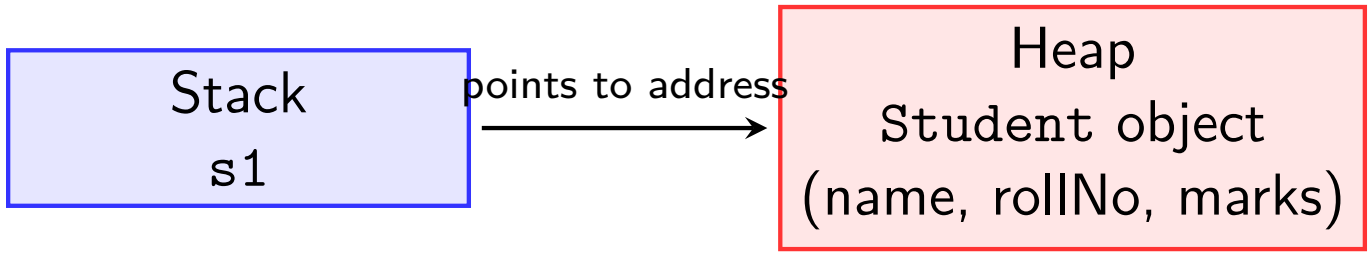
- JVM memory areas: stack vs heap
- Object references: what a variable actually holds
- Aliasing: multiple references, one object
- Garbage collection: reclaiming unused objects

Where Do Objects Live? Stack vs Heap

- Recall Lecture 1's JVM architecture: the JVM manages several memory areas at runtime
- The heap is where all objects (created with `new`) actually reside
- The stack holds method call frames, including local variables and reference variables
- So a reference variable like s1 lives on the stack, while the Student object it points to lives on the heap

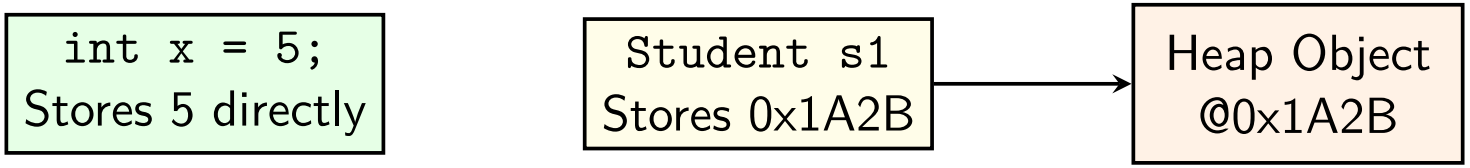
Stack Frame and Heap Object

- `Student s1 = new Student();`
- Stack: frame for main() contains the variable `s1`
- `s1` does not hold the object itself - it holds the heap address of the object
- Heap: the actual Student object (with its name, rollNo, marks fields) is stored here



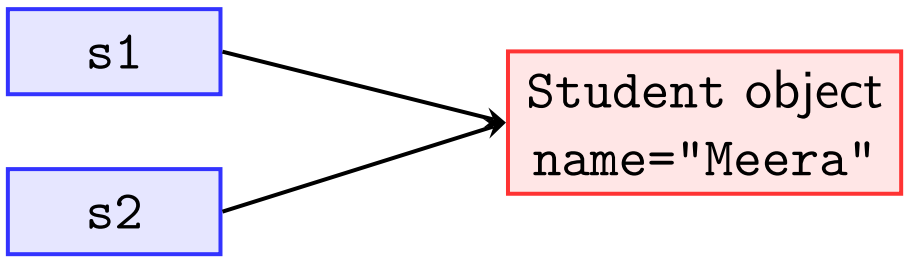
What a Reference Variable Actually Stores

- A reference variable does not store the object's fields directly - it stores the object's memory address
- Primitive variables directly store their value on the stack
- Reference variables store an address pointing to heap data
- Assigning one object variable to another copies the address, not the object



Aliasing: Two References, One Object

```
Student s1 = new Student();
s1.name = "Asha";
Student s2 = s1;    // s2 now holds the SAME address as s1
s2.name = "Meera";
System.out.println(s1.name);  // prints Meera, not Asha!
```



Aliasing vs. Two Separate Objects

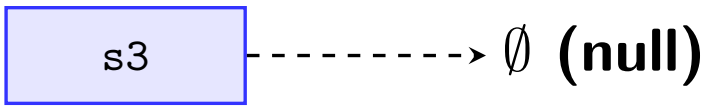
Two Separate Objects	Aliasing
`Student s2 = new Student();`	`Student s2 = s1;`
Creates a brand-new, independent object on the heap	No new object; s2 is just another name (alias) for the same object

Rule of thumb

Only `new` creates a new object; plain assignment between object variables just copies a reference.

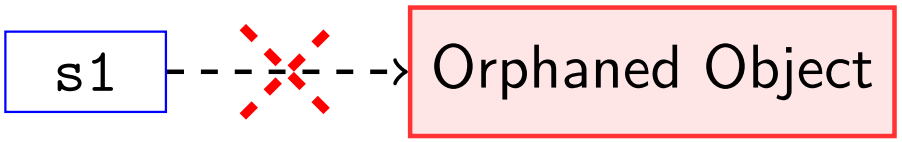
Null References

```
Student s3 = null;     // s3 points to nothing
s3.displayInfo();      // throws NullPointerException at runtime
```



When Does an Object Become Unreachable?

- An object becomes unreachable when no reference variable anywhere in the program still points to it
- Example: `s1 = null;` (if s1 was the only reference to that object) makes the old object unreachable
- Example: reassigning `s1 = new Student();` to a new object also abandons the previous object
- Unreachable objects still occupy heap memory until they are cleaned up



Garbage Collection: Automatic Memory Management

- Garbage collection (GC) is the JVM's automatic process of finding and reclaiming unreachable objects' memory
- Programmers never call a 'free' or 'delete' function in Java, unlike languages such as C or C++
- The JVM runs the garbage collector periodically in the background, in a way largely invisible to the running program
- This is a major reason Java programs are less prone to memory leaks and dangling-pointer bugs than manually managed languages

How the Garbage Collector Works (Conceptual)

- Conceptually, the GC periodically scans the heap starting from all currently reachable references
- Any object it cannot reach through this scan is considered garbage and its memory is reclaimed
- The exact algorithm is an implementation detail of the JVM, not required at this syllabus depth
- Students only need to know: 'no references $\rightarrow$ eligible for garbage collection'

Java's Automatic GC vs. Manual Memory Management

Java (Automatic)	**C/C++ (Manual)**
Objects are automatically reclaimed by the garbage collector once unreachable	The programmer must explicitly `free()` / `delete` allocated memory
No explicit deallocation code needed	Permanent memory leaks if omitted
Less manual memory-bug risk, but slightly less direct control over exactly when memory is freed	Complete control over deallocation time, but high risk of human error

Stack vs Heap: Summary Table

Memory Area	What it stores	How it is reclaimed
Stack	Method frames, local variables, and reference variable values (addresses)	Automatically when a method returns
Heap	All objects created with `new`, including their instance variables	By the garbage collector once an object becomes unreachable

Key Takeaways

- Objects are stored in heap memory; reference variables stored on the stack point to those heap objects
- A reference variable holds an object's memory address, not the object itself - so multiple references can alias the same object
- When no reference anywhere in the program points to an object, it becomes eligible for garbage collection
- Java automates memory reclamation via the garbage collector, unlike languages that require manual deallocation

Next Lecture Preview

- Lecture 13 shifts to encapsulation's enforcement mechanism: the four access modifiers (public, private, protected, default) that control visibility of class members.

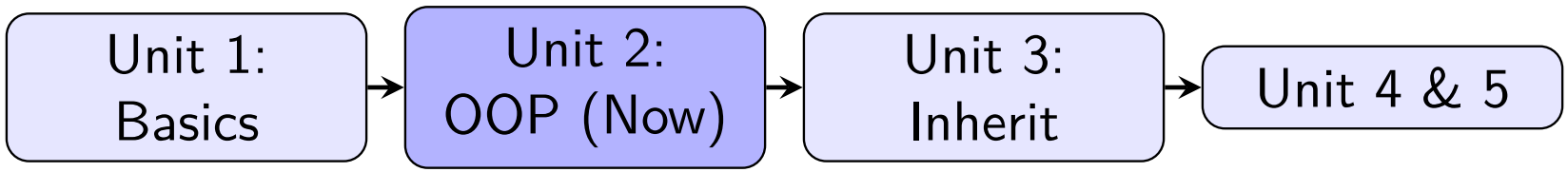
Questions?

Recap: Memory, References, and Garbage Collection

- Objects live on the heap; reference variables on the stack point to them
- Assigning one reference to another creates an alias to the SAME object, not a copy
- Objects with no remaining references become eligible for garbage collection

Course Overview & Today's Agenda

- Why control access to class members?
- The four access modifiers: public, private, protected, default
- Visibility scope comparison across package and subclass boundaries
- Choosing the right modifier for fields vs methods

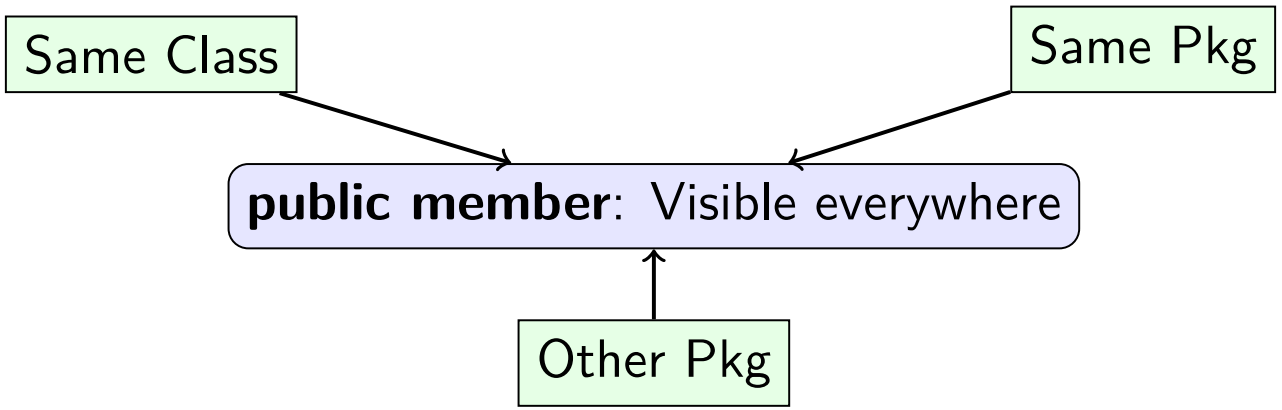


Why Restrict Access to Class Members?

- So far, our Student fields have been directly accessible: `s1.marks = -50;` compiles fine
- Uncontrolled access lets any code corrupt an object's data with invalid values
- Access modifiers let a class control exactly who sees/changes members
- This enables true encapsulation (completed with getters/setters in Lecture 14)

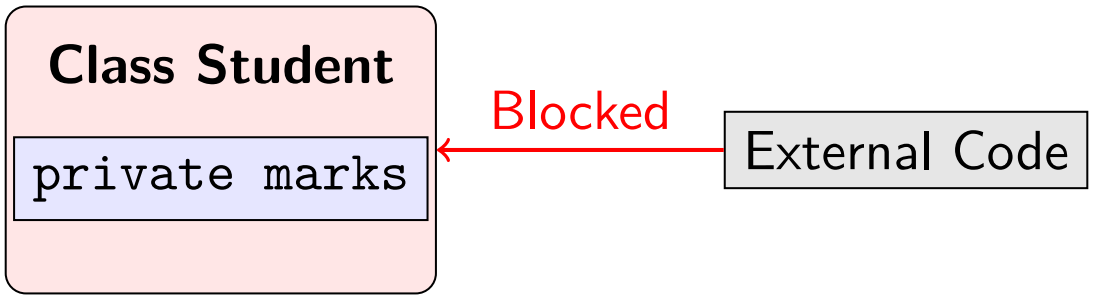
public: Accessible Everywhere

```java
public class Student {
    public String name;
}
// From ANY class, in ANY package:
Student s1 = new Student();
s1.name = "Asha";    // allowed - name is public
```



private: Accessible Only Within the Class

```java
public class Student {
    private int marks;
}

Student s1 = new Student();
s1.marks = 88;   // COMPILE ERROR: marks has private access
```



protected: Package Plus Subclasses

```java
public class Student {
    protected int marks;
}
// Accessible: same class, same package,
// AND subclasses in OTHER packages too
```

Same Package
(Accessible)

Subclass in Other Pkg
(Accessible)

Other Pkg Non-Subclass
(Blocked)

default (Package-Private): No Modifier Written

```java
class Student {
    int marks;   // no modifier = default access
}
// Accessible: any class in the SAME package only
// NOT accessible from a class in a different package
```

Same Package
All classes can access

Other Packages
Blocked

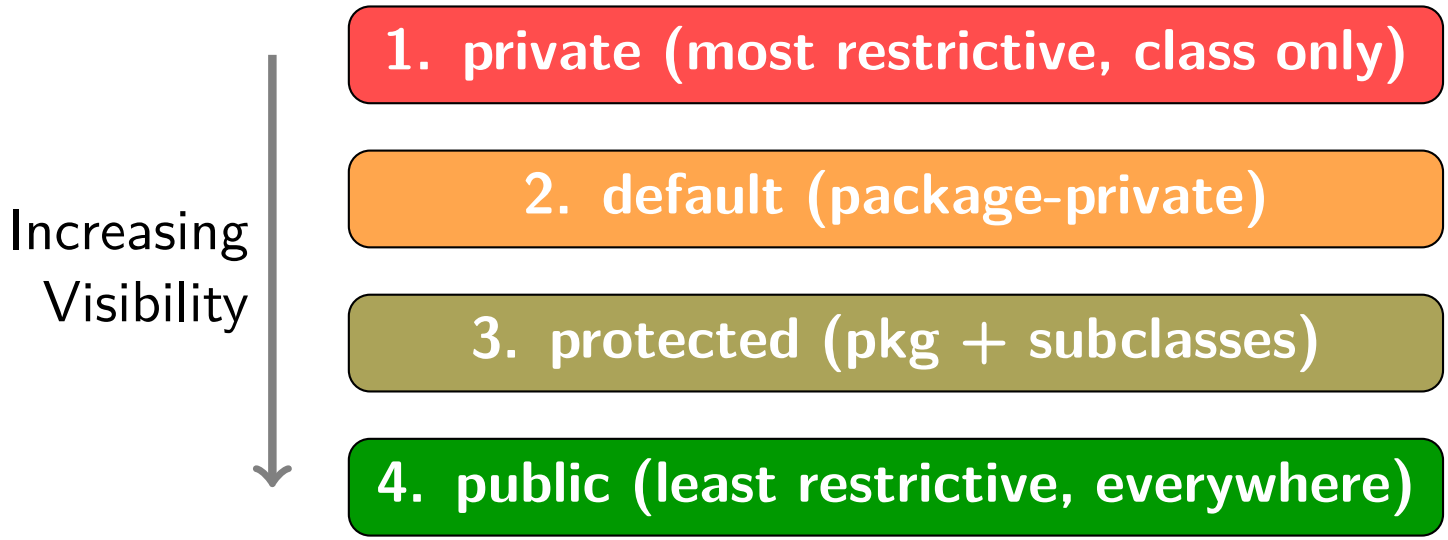
Access Modifier Visibility Matrix

Context	public	protected	default	private
Same class	Yes	Yes	Yes	Yes
Same package	Yes	Yes	Yes	No
Subclass (different pkg)	Yes	Yes	No	No
Different pkg	Yes	No	No	No

One Class, Four Modifiers Demonstrated

```
public class Demo {
    public int a;       // accessible everywhere
    protected int b;    // package + subclasses
    int c;              // default: package only
    private int d;      // this class only
}
```

Ranking the Modifiers by Restrictiveness



Choosing the Right Modifier

- Declare fields **private** to protect an object's internal state
- Declare methods for the external API as **public**
- Use **protected** for intended subclass access
- Use **default** sparingly for package-internal helpers

Encapsulation Rule of Thumb

Fields should be `private`, and accessed only via `public` methods.

Common Mistakes with Access Modifiers

- Leaving fields as public or default (allows uncontrolled modification)
- Forgetting that private members are invisible even to subclasses
- Assuming protected is like private in the same package (it's as open as default)
- Applying access modifiers to local variables inside a method (not allowed)

Key Takeaways

- Java provides four access modifiers: `public`, `private`, `protected`, `default`
- `private` is the most restrictive; `public` is the least restrictive
- `protected` extends default to include subclasses in other packages
- Best practice: declare fields `private` and expose via `public` methods

Next Lecture Preview

- Lecture 14 builds directly on private fields
- Implementing full data hiding through getter and setter methods
- Completing the encapsulation principle

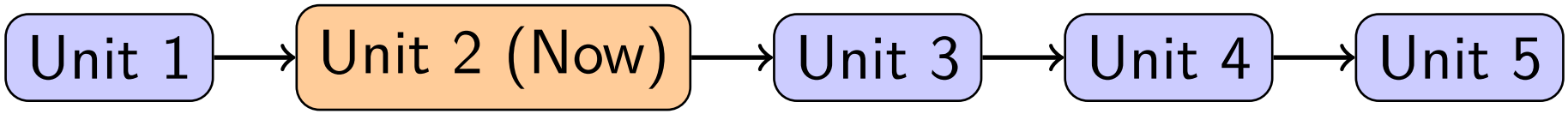
Questions?

Recap: Access Modifiers

- Java has four access modifiers: private, default, protected, public - from most to least restrictive
- private restricts a member to its own class only
- Best practice guideline introduced: fields private, methods public

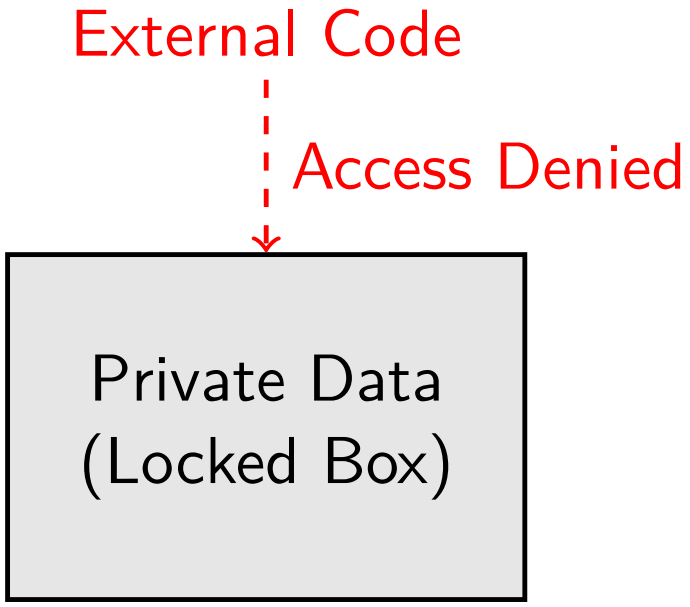
Course Overview & Today's Agenda

- Data hiding: keeping fields private
- Why we need getters and setters
- Writing getter methods
- Writing setter methods with validation logic



Data Hiding: The Principle

- Data hiding: internal fields are not directly visible or modifiable from outside
- Implements the encapsulation pillar introduced in Lecture 10
- Achieved by declaring fields private
- The class must still provide a controlled way to read and update the data



The Problem: private Alone Isn't Enough

```
public class Student {
    private int marks;
}

Student s1 = new Student();
s1.marks = 88;      // COMPILE ERROR - marks is private
System.out.println(s1.marks);  // COMPILE ERROR too
```

Getter Methods: Reading Private Data

```
public class Student {
    private int marks;

    public int getMarks() {
        return marks;
    }
}

System.out.println(s1.getMarks());   // works!
```

Setter Methods: Controlled Writing

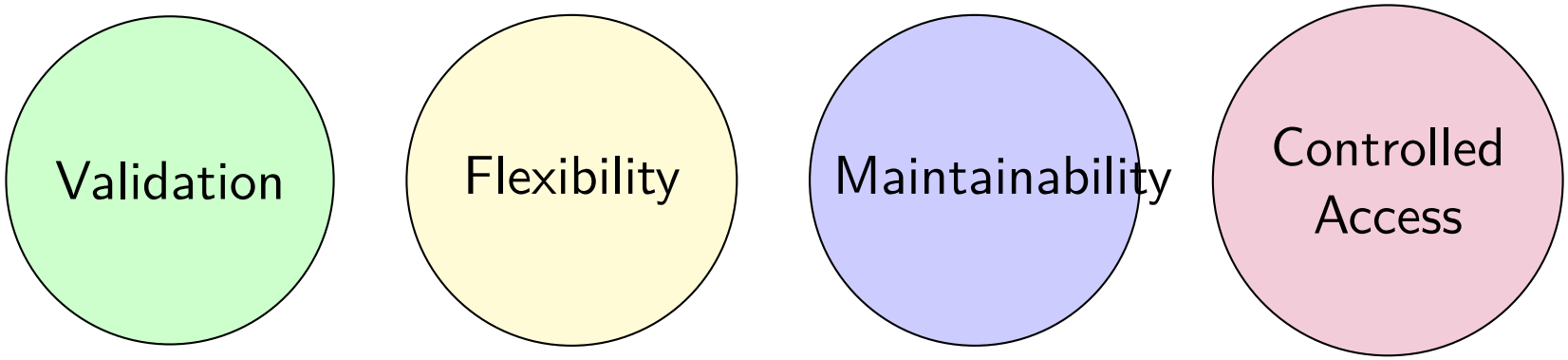
```
public class Student {
    private int marks;

    public void setMarks(int marks) {
        if (marks >= 0 && marks <= 100) {
            this.marks = marks;
        } else {
            System.out.println("Invalid marks!");
        }
    }
}
```

A Fully Encapsulated Student Class

```
public class Student {
    private String name;
    private int marks;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getMarks() { return marks; }
    public void setMarks(int marks) {
        if (marks >= 0 && marks <= 100) this.marks = marks;
    }
}
```

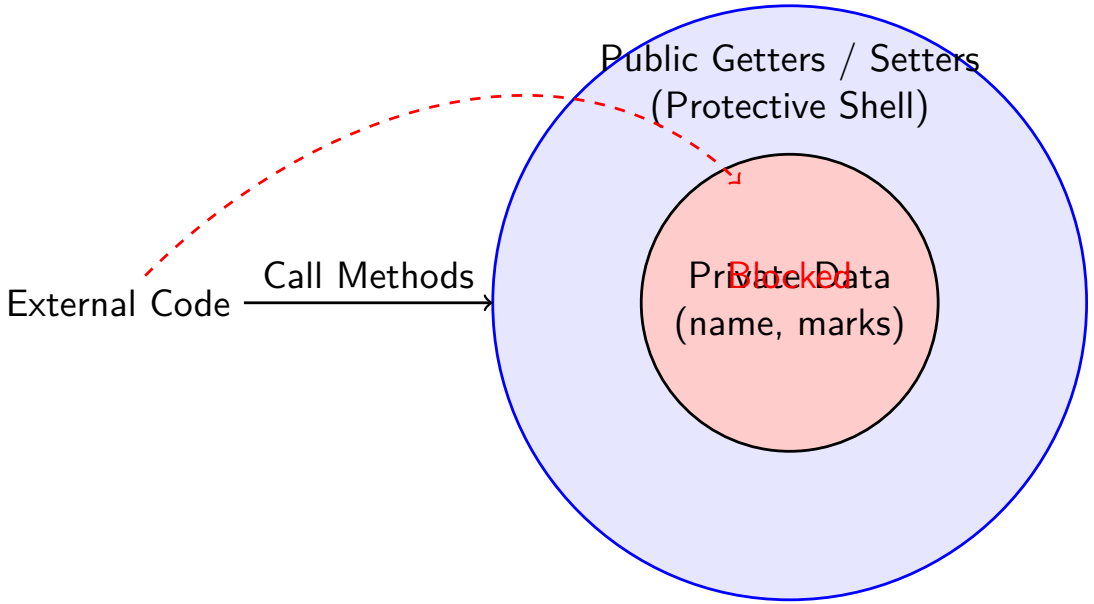
Why Encapsulation Matters: Benefits

- Validation: setters can reject invalid data before it corrupts the object
- Flexibility: internal representation can change later without breaking external code
- Maintainability: reading/writing logic lives in one place
- Controlled access: omit setter or getter for read-only or write-only



Encapsulation as a Capsule

- Private data sits at the center, inaccessible directly from outside
- Public getter/setter methods form a protective shell around the private data
- All outside interaction must pass through this shell - direct access to the center is blocked



Naming Convention for Accessors

- Getter for a field named `x` of type `T`: `public T getX() { return x; }`
- Setter for a field named `x`: `public void setX(T x) { this.x = x; }`
- Special case: for a boolean field, getter is conventionally named `isX()`
- Following convention makes classes predictable for Java tools and frameworks

Naming Pattern Reference

```
private Type fieldName;
public Type getFieldName() { ... }
public void setFieldName(Type value) { ... }
```

Direct Field Access vs. Encapsulated Access

Direct Public Field	**Encapsulated Field**
`s1.marks = -50;` compiles and silently corrupts data	`s1.setMarks(-50);` detects and rejects the invalid value
Any future change to type/name breaks external code	Internal representation can change freely without breaking code

Common Mistakes with Getters and Setters

- Forgetting to add validation logic in the setter, defeating the purpose of encapsulation
- Making the getter or setter private, so no outside code can call it at all
- Writing `marks = marks;` in a setter instead of `this.marks = marks;` when the parameter shadows the field
- Adding getters/setters for every single field automatically, even when a field should be read-only

Key Takeaways

- Data hiding keeps fields private and exposes controlled access only through public methods
- Getter methods return field values; setter methods assign values, ideally after validating input
- The standard Java naming convention is getFieldName()/setFieldName() (or isFieldName() for booleans)
- Encapsulation via private fields plus public getters/setters protects data integrity and allows internal changes

Next Lecture Preview

- Lecture 15 turns to methods themselves in more depth - how parameters are passed in Java, and the `this` keyword that resolves naming conflicts between fields and parameters.

Questions?

Recap: Getters and Setters

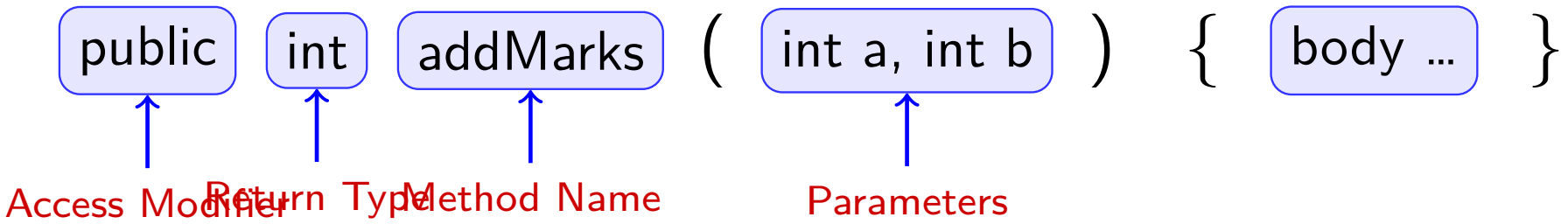
- Getter methods return a private field's value; setter methods assign it, often with validation
- A setter like `setMarks(int marks)` already uses parameters and `this.marks = marks;`
- Today we look closely at methods and `this` in general, beyond just getters and setters

Course Overview & Today's Agenda

- Method anatomy: return type, name, parameters, body
- How Java passes arguments: pass-by-value for primitives and references
- The `this` keyword: referring to the current object
- Using `this` to resolve field/parameter name clashes

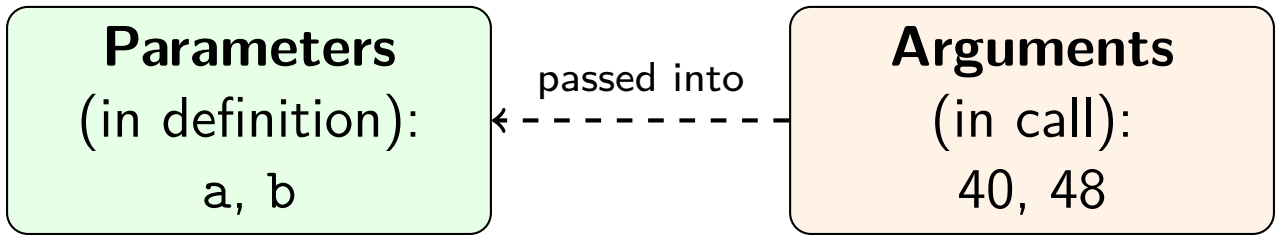
Anatomy of a Java Method

- General form: `accessModifier returnType methodName(parameterList) { body }`
- Return type: the data type of the value the method sends back, or `void` if it returns nothing
- Parameters: variables listed in parentheses that receive values when the method is called
- Method body: the block of statements executed when the method runs



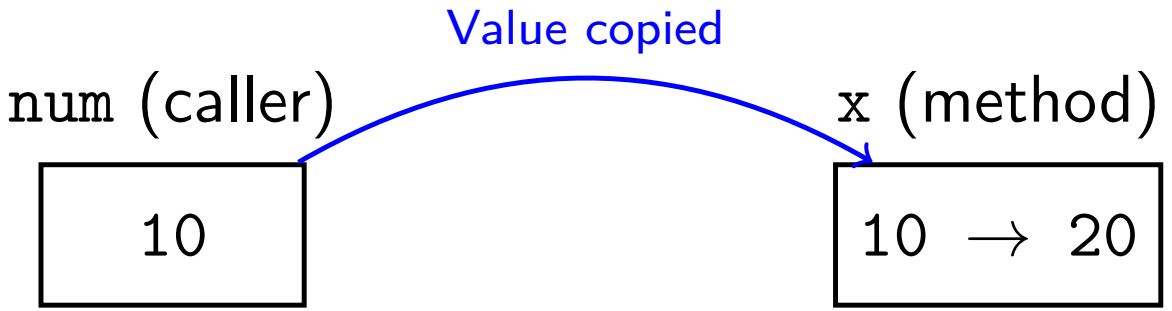
Defining and Calling a Method with Parameters

```java
public int addMarks(int a, int b) {
    int total = a + b;
    return total;
}

int result = s1.addMarks(40, 48);  // result = 88
```



Pass-by-Value: Primitives

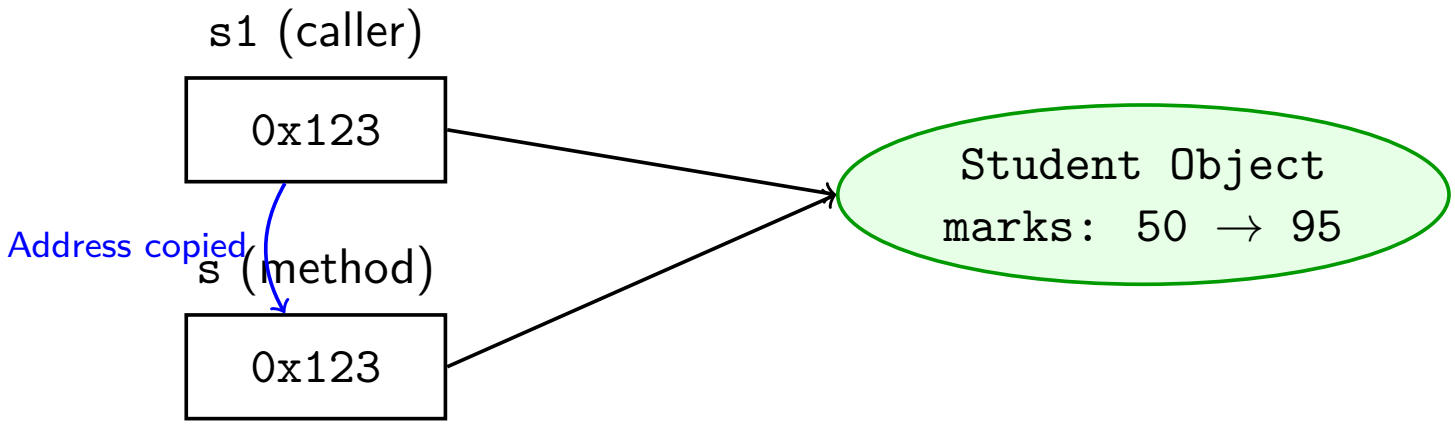
```
public void doubleIt(int x) {
    x = x * 2;
}

int num = 10;
doubleIt(num);
System.out.println(num);  // still prints 10, NOT 20
```



Changing `x` does not affect `num`

Pass-by-Value: Object References

```
public void updateMarks(Student s) {
    s.setMarks(95);    // mutates the SAME object
}

Student s1 = new Student();
s1.setMarks(50);
updateMarks(s1);
System.out.println(s1.getMarks());  // prints 95
```



What Changes Persist After a Method Call?

Action inside method	Example Code	Effect on Caller
Reassign primitive parameter	`x = x * 2;`	LOCAL ONLY. Caller's variable is completely unaffected.
Change object's FIELDS	`s.setMarks(95);`	VISIBLE. The same shared object is mutated in memory.
Reassign object reference	`s = new Student();`	LOCAL ONLY. Method now points to a new object, caller still points to old object.

The Problem: Field and Parameter Name Clash

```
public class Student {
    private String name;

    public void setName(String name) {
        name = name;    // does NOTHING useful!
    }
}
```

Self-Assignment Bug

The parameter `name` shadows the field `name`. The assignment just assigns the parameter to itself, leaving the field unchanged.

The this Keyword: Referring to the Current Object

```
public class Student {
    private String name;

    public void setName(String name) {
        this.name = name;  // this.name = the field
    }
}
```

this Keyword

this is a reference to the current object on which a method is invoked. It unambiguously identifies instance fields when shadowed by parameters.

Other Common Uses of `this`

- Resolving field/parameter name clashes in setters and constructors (just seen)
- Invoking another constructor of the same class using `this(...)` - covered fully in Lecture 16's constructor chaining
- Passing the current object as an argument to another method, e.g., `someMethod(this)`
- `this` can only be used inside instance methods and constructors - never inside a static method (previewed further in Lecture 18)

A Complete Setter Using this

```
public class Student {
    private String name;
    private int marks;

    public void setName(String name) {
        this.name = name;
    }
    public void setMarks(int marks) {
        if (marks >= 0 && marks <= 100) {
            this.marks = marks;
        }
    }
}
```

Common Mistakes with Methods and `this`

- Forgetting `this` when a parameter name matches a field name, causing a silent no-op bug
- Expecting an object argument's reassignment inside a method to affect the caller's variable (it never does)
- Trying to use `this` inside a static method or static context, which does not compile (there is no current object)
- Confusing 'parameter' (in the method definition) with 'argument' (in the method call) when discussing exam answers

Key Takeaways

- A Java method has a return type, name, parameter list, and body; use `void` when no value is returned
- Java always passes arguments by value - for object references, the reference's value (the address) is copied, so the same object can still be mutated through the parameter
- The `this` keyword refers to the current object and resolves naming conflicts between instance fields and parameters
- Omitting `this` in a setter where the parameter shadows the field is a very common bug that silently does nothing

Next Lecture Preview

- Lecture 16 covers constructors in depth - default, parameterized, and copy constructors, plus constructor chaining using `this()`.

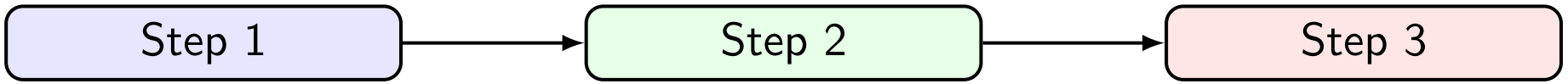
Questions?

Recap: Methods and ‘this‘

- Java methods have a return type, name, parameters, and body; arguments are always passed by value
- The ‘this‘ keyword refers to the current object and resolves field/parameter name clashes
- ‘this‘ can also be used to call another constructor of the same class - today’s main topic

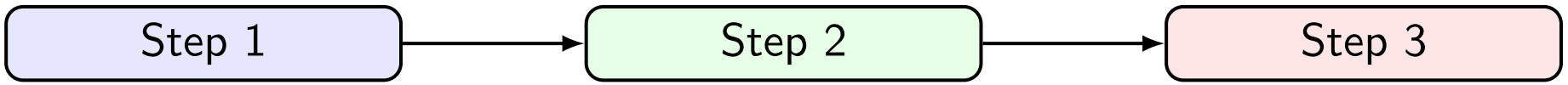
Course Overview & Today's Agenda

- What is a constructor? How it differs from methods
- Default constructor (no-argument)



What Is a Constructor?

- A constructor is a special block of code that runs automatically when an object is created with ‘new‘
- It shares the exact same name as the class, and has NO return type - not even ‘void‘



Constructor vs. Regular Method

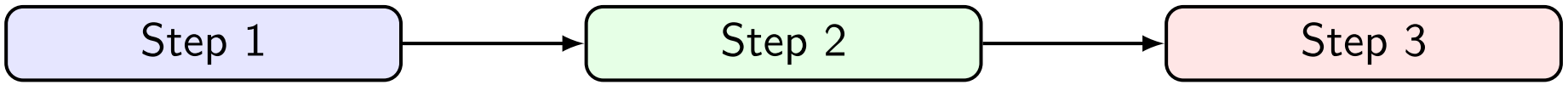
Aspect	Constructor	Method
Name	{MUST} match the class name exactly	can have any valid identifier name
Return type	has NO return type (not even void)	must declare one (or void)
Invocation	runs automatically and only via ‘new‘	must be called explicitly using the dot operator
Purpose	initializes a new object’s state	implements ongoing behavior

The Default (No-Argument) Constructor

```
public class Student {
    String name;
    int marks;

    public Student() {
        name = "Unknown";
        marks = 0;
        System.out.println("Student object created");
    }
}
```

The Compiler-Provided Default Constructor

- If a class defines NO constructor at all, Java automatically supplies an invisible, empty no-argument constructor
- This is exactly what has been happening in every Student example since Lecture 11 - Java quietly provided one for us



Parameterized Constructor

```
public class Student {
    String name;
    int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }
}

Student s1 = new Student("Asha", 88);
```

Copy Constructor

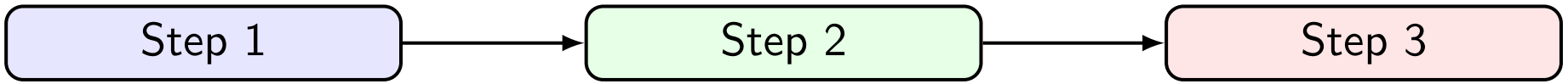
```
public class Student {
    String name; int marks;

    public Student(Student other) {
        this.name = other.name;
        this.marks = other.marks;
    }
}

Student s2 = new Student(s1);  // copies s1's data into s2
```

Constructor Chaining with this()

```
public class Student {
    String name; int marks;

    public Student() {
        this("Unknown", 0);    // calls the parameterized
            constructor
    }
    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }
}
```

Constructor Chaining: Call Flow and Rules

- ‘new Student()‘ is called -> the no-argument constructor body starts running
- Its first statement, this(”Unknown”, 0), immediately transfers control to the parameterized constructor



Rules for Writing Constructors

- The constructor name must exactly match the class name (including capitalization)
- A constructor must not declare any return type, not even void - adding one turns it into a regular method by mistake
- If used, this(...) must be the very first statement inside the constructor body
- A class can have multiple constructors as long as their parameter lists differ (this is constructor overloading, covered fully in Lecture 17)

Common Mistakes with Constructors

- Accidentally adding a return type (e.g., ‘void Student()‘), which silently turns the constructor into an ordinary method with the same name as the class
- Writing a parameterized constructor and then being surprised that ‘new Student()‘ no longer compiles
- Placing a statement before this(...) inside a constructor, which is a compile error
- Confusing a copy constructor (Student(Student other)) with plain reference assignment (Student s2 = s1;), which only creates an alias, not a copy

Key Takeaways

- A constructor shares the class's name, has no return type, and runs automatically when an object is created with ‘new‘
- Java supplies a default no-argument constructor only if you define no constructors yourself
- Parameterized and copy constructors let objects be initialized with specific values, or duplicated from an existing object, at creation time
- Constructor chaining via ‘this(...)‘ (which must be the first statement) lets one constructor reuse another's initialization logic, avoiding duplication

Next Lecture Preview

- Lecture 17 introduces method and constructor overloading - defining multiple versions of the same method or constructor name that differ in their parameter lists.

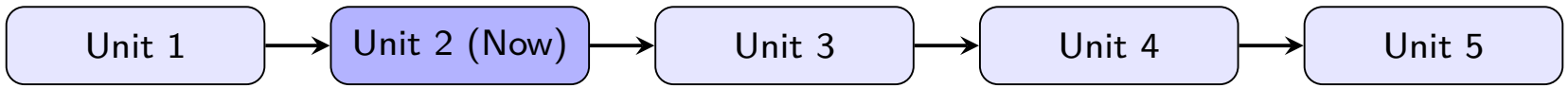
Questions?

Recap: Constructor Types and Chaining

- Default, parameterized, and copy constructors initialize objects in different ways
- The compiler only supplies a default constructor if the class defines none of its own
- `this(...)` chains one constructor to call another within the same class

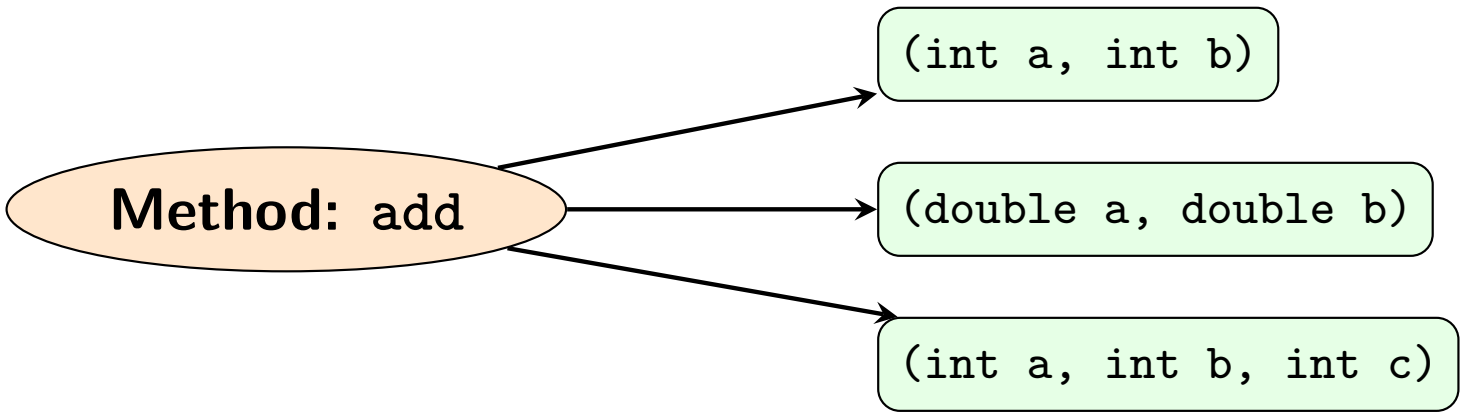
Course Overview & Today's Agenda

- What is method overloading and why use it?
- Valid overloading: differing by number, type, or order of parameters
- Compile-time resolution of overloaded calls
- Constructor overloading: multiple ways to build an object



What Is Method Overloading?

- Method overloading means defining two or more methods with the SAME name in the same class
- The methods must differ in their parameter list - the number, type, or order of parameters
- Purpose: lets one intuitive method name (e.g., add) work sensibly with different kinds or numbers of inputs



Overloading in Action

```
public int add(int a, int b) {
    return a + b;
}
public double add(double a, double b) {
    return a + b;
}
public int add(int a, int b, int c) {
    return a + b + c;
}
```

Valid Ways to Overload a Method

- Differ by number of parameters: `add(int, int)` vs `add(int, int, int)`
- Differ by parameter type: `add(int, int)` vs `add(double, double)`
- Differ by parameter order (when types differ): `display(String, int)` vs `display(int, String)`

Key Rule

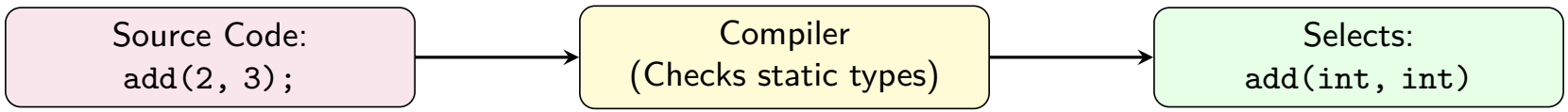
Any one of these differences alone is sufficient to make an overload valid.

What Does NOT Count as Overloading

```java
public int add(int a, int b) { return a + b; }
public double add(int a, int b) { return a + b; }
// COMPILE ERROR: methods have the SAME signature
// (name + parameter list) - only return type differs!
```

Compile-Time Resolution: Static Polymorphism

- The Java compiler examines the arguments' types and picks the matching overload
- This decision is made at COMPILE time, based purely on the static types of the arguments
- Because the compiler decides, method overloading is called compile-time or static polymorphism



Tracing Overloaded Calls

```
add(2, 3);          // calls add(int, int)          -> 5
add(2.5, 3.5);      // calls add(double, double)    -> 6.0
add(1, 2, 3);       // calls add(int, int, int)     -> 6
```

Constructor Overloading

```
public class Student {
    String name; int rollNo; int marks;

    public Student() {
        this("Unknown", 0, 0);
    }
    public Student(String name) {
        this(name, 0, 0);
    }
    public Student(String name, int rollNo, int marks) {
        this.name = name; this.rollNo = rollNo; this.marks =
            marks;
    }
}
```

Constructor Overloading Gives Flexible Object Creation

```
Student s1 = new Student();                    // all defaults
Student s2 = new Student("Ravi");              // name only
Student s3 = new Student("Asha", 21, 88);      // fully specified
```

Method Overloading vs. Constructor Overloading

Method Overloading	**Constructor Overloading**
Multiple methods, same name, different parameter lists	Multiple constructors, name fixed to class name, different parameter lists
Resolved at compile time based on argument types (static polymorphism)	Resolved at compile time based on argument types (static polymorphism)
Can have any return type	No return type, used to initialize objects
Can use `this.field = param;`	Commonly use `this(...)` to avoid duplicated logic

Common Mistakes with Overloading

- Trying to overload using only a different return type, which does not compile
- Creating ambiguous overloads, e.g., calling `add(5, 5)` when both `add(int,long)` and `add(long,int)` exist (ambiguity error)
- Forgetting that defining any constructor removes the automatic no-argument one
- Confusing method overloading (compile-time) with method overriding (runtime)

Key Takeaways

- Method overloading means defining multiple methods with the same name in a class, distinguished by parameter number, type, or order
- Return type alone cannot distinguish overloaded methods - the parameter list must differ
- The compiler resolves which overload to invoke at compile time based on argument types
- Constructor overloading provides multiple ways to initialize an object and commonly chains via `this()`

Next Lecture Preview

- Lecture 18 completes Unit 2 with static variables, methods, and blocks, plus the final keyword - and wraps up the entire unit before Inheritance begins in Unit 3.

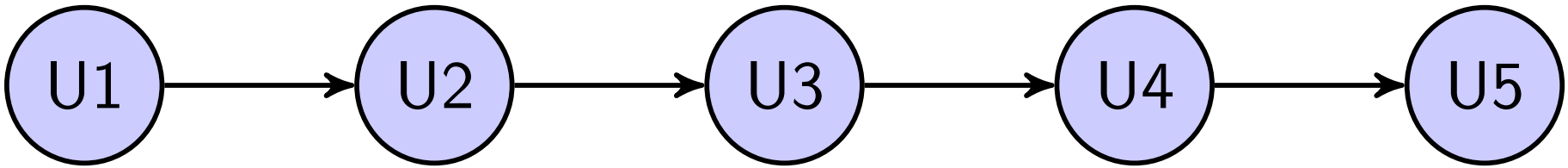
Questions?

Recap: Method and Constructor Overloading

- Method overloading: same method name, different parameter lists, resolved at compile time
- Constructor overloading applies the same idea to constructors, often chained via this(...)
- Both are examples of compile-time (static) polymorphism

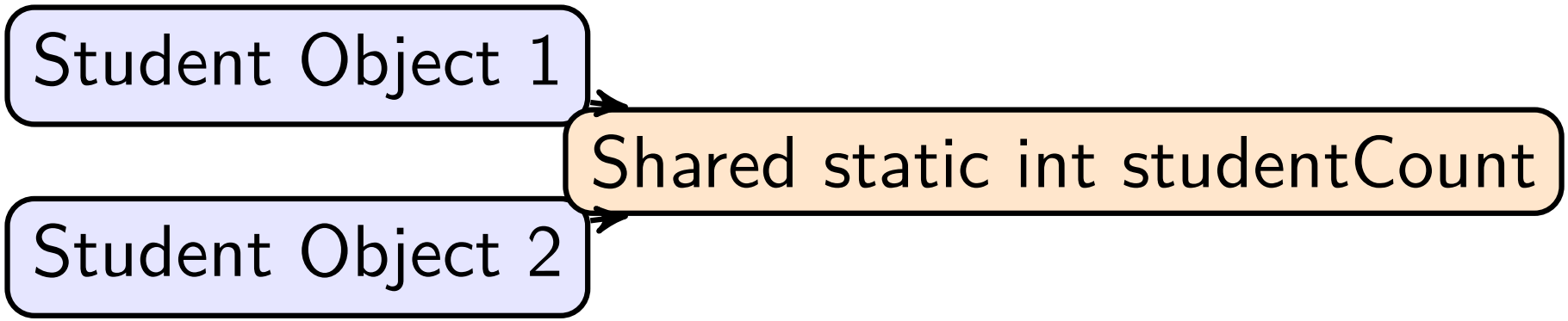
Course Overview & Today's Agenda

- Static variables: one copy shared by all objects
- Static methods, restrictions, and static blocks
- The final keyword: variables, methods, and classes



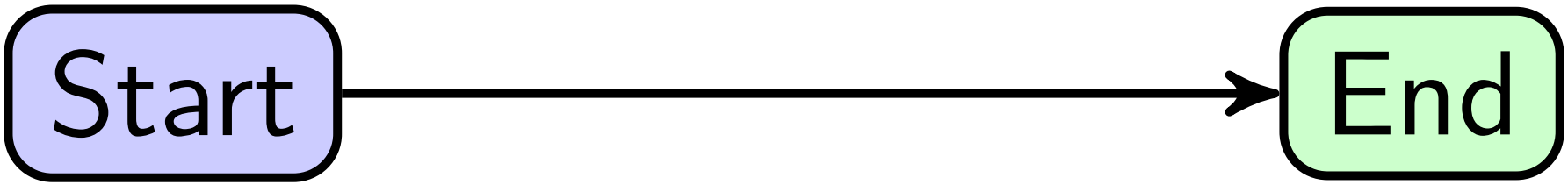
Static Variables: Shared Across All Objects

```
public class Student {
    String name;
    static int studentCount = 0;    // ONE copy shared by all
        objects

    public Student(String name) {
        this.name = name;
        studentCount++;
    }
}
```



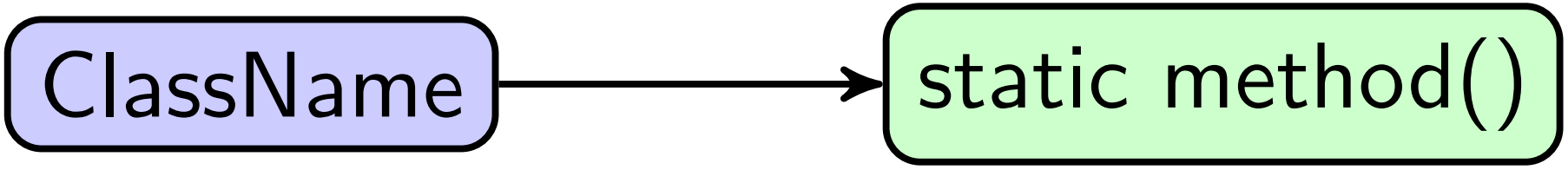
Instance Count Demonstration

- Student s1 = new Student("Asha");
- Student s2 = new Student("Ravi");
- Student s3 = new Student("Meera");



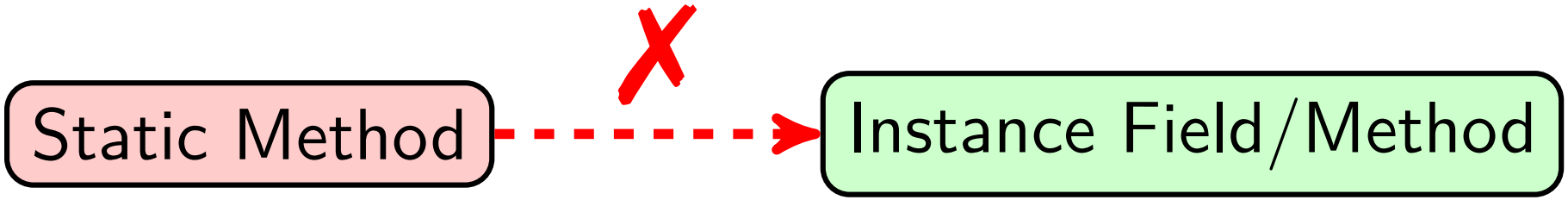
Static Methods

```
public class Student {
    static int studentCount = 0;

    public static int getStudentCount() {
        return studentCount;    // OK: static method + static
            field
    }
}

System.out.println(Student.getStudentCount());
```



Restrictions on Static Methods

- A static method cannot directly access instance (non-static) fields or call instance methods
- Reason: instance members belong to a specific object, but a static method may run with no object involved at all
- A static method has no ‘this‘ reference - there is no ’current object’ inside a static context



Instance Members vs. Static Members

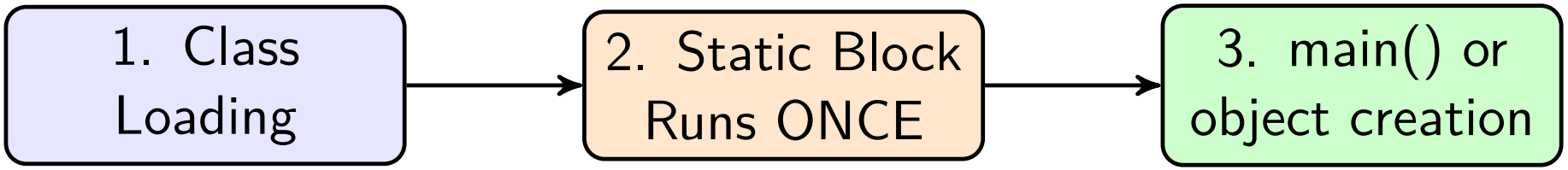
Topic / Concept	Detail
Instance variable	one copy per object (e.g., each Student's own name and marks)
Static variable	one copy per class, shared by every object (e.g., studentCount)
Instance method	called on a specific object (s1.displayInfo()), can use 'this'
Static method	called on the class itself (Student.getStudentCount()), cannot use 'this'

Static Blocks

```
public class Config {
    static String appName;

    static {
        appName = "StudentPortal";
        System.out.println("Static block executed");
    }
}
```

When Does a Static Block Run?

- 1. JVM loads the Config class (the first time it is referenced)
- 2. All static blocks and static field initializers run, in the order they appear, exactly ONCE
- 3. This happens BEFORE main() runs (if the static block is in the same class as main) and before any object of the class is created



The final Keyword: Variables

```
final int MAX_MARKS = 100;
MAX_MARKS = 90;    // COMPILE ERROR: cannot reassign a final
    variable

final Student topper = new Student("Asha");
topper.setMarks(95);          // OK - the OBJECT's data can still
    change
topper = new Student("Ravi");  // COMPILE ERROR - reference
    itself is final
```

The final Keyword: Methods and Classes

- final method: cannot be overridden by any subclass - this directly previews method overriding in Unit 3, Lecture 20
- final class: cannot be extended/subclassed at all - this directly previews inheritance and the ‘extends‘ keyword starting Lecture 19
- Common real example: the Java String class itself is declared final, so no one can create a subclass of String
- Using final on a method or class is a deliberate design decision to prevent further extension or modification

final Applied to Variables, Methods, and Classes

- `final` variable - value can be assigned only once; reassignment is a compile error
- `final` method - cannot be overridden by a subclass (relevant once inheritance begins, Unit 3)
- `final class` - cannot be extended/subclassed at all (relevant once inheritance begins, Unit 3)
- Common use: `final` is often combined with `static` to declare class-wide constants, e.g., `static final int MAX_MARKS = 100;`

Unit 2 Wrap-Up: Topics 2.1 to 2.5

Topic / Concept	Detail
2.1 (Lecture 10): OOP fundamentals	class, object, encapsulation, inheritance, polymorphism; procedural vs OOP
2.2 (Lectures 11 & 12): defining classes	creating objects with ‘new‘, memory allocation, references, and garbage collection
2.3 (Lectures 13 & 14): access modifiers	data hiding, and getter/setter methods
2.4 (Lectures 15 & 16): methods	pass-by-value parameter passing, the ‘this‘ keyword, and constructor types/chaining
2.5 (Lecture 17 & 18): method/constructor overloading	static variables/methods/blocks, and the final keyword

Common Mistakes with static and final

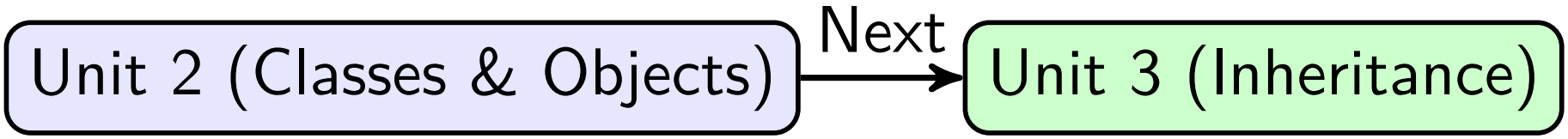
- Trying to access an instance (non-static) field or method directly from inside a static method
- Forgetting that a static block runs only once per class loading, not once per object created
- Believing ‘final‘ on an object reference freezes the object’s internal state too - it only prevents reassigning the reference itself
- Confusing ‘final‘ (this lecture, prevents change/override/extension) with ‘finally‘ (exception handling, Unit 4) - different keywords with different purposes

Key Takeaways

- Static variables belong to the class itself (one shared copy), while instance variables belong to each object separately
- Static methods are invoked via the class name, cannot directly access instance (non-static) members, and have no ‘this‘ reference
- A static block runs exactly once, when the class is first loaded by the JVM, and is typically used to initialize static fields
- The final keyword locks a variable’s value, prevents a method from being overridden, and prevents a class from being extended - previewing inheritance in Unit 3

Next Lecture Preview

- Lecture 19 begins Unit 3 (Inheritance, Interfaces, and Packages) with inheritance concepts and the ‘extends‘ keyword, building directly on the classes and objects mastered throughout Unit 2.



Questions?

Recap: Closing Out Unit 2

- **Lecture 18**: `static` variables/methods/blocks - shared across all objects of a class
- **Lecture 18**: `final` keyword - locks variables (constants), methods (no override), and classes (no subclassing)
- **Unit 2 overall**: OOP pillars, classes/objects/memory, access modifiers/encapsulation, constructors/`this`, overloading

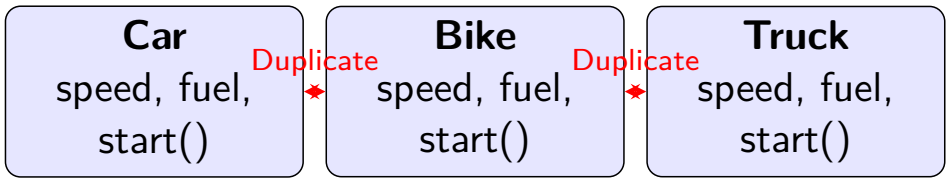
Course Overview & Today's Agenda

- **Unit 3 roadmap**: Inheritance (L19-20), Overriding & Object class (L21-22), Interfaces (L23-24), Abstract classes (L25), Packages (L26-27)
- **Today**: what inheritance is, why we need it, and the IS-A relationship
- Superclass vs subclass terminology and the `extends` keyword syntax
- First hands-on example: Animal $\rightarrow$ Dog, and what is/isn't inherited

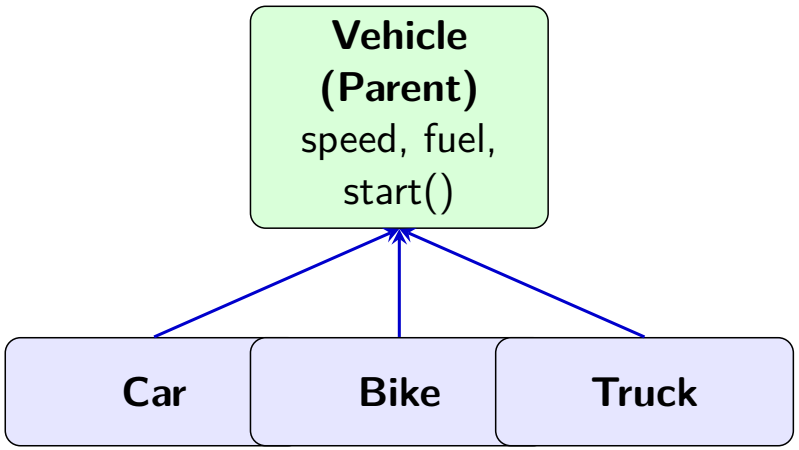
Why Do We Need Inheritance?

- Real programs often share common fields/methods with small differences
- Without inheritance, we copy-paste code - hard to maintain
- Inheritance reuses fields/methods of an existing class

Without Inheritance

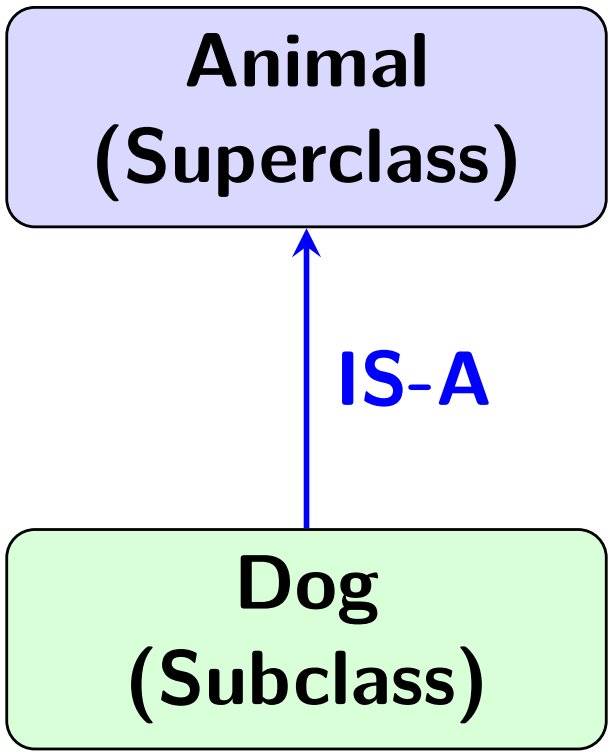


With Inheritance



Inheritance: Definition and Terminology

- **Inheritance**: new class acquires fields/methods of an existing class
- **Superclass** (parent): the existing class being inherited from
- **Subclass** (child): the new class that inherits from the superclass
- Creates an **IS-A relationship**: a Dog IS-A Animal, a Car IS-A Vehicle



The extends Keyword: Syntax

- The extends keyword goes directly after the subclass name
- It creates the inheritance relationship automatically

```
class Animal {
    String name;
    void eat() { System.out.println(name + " is eating"); }
}

class Dog extends Animal {
    void bark() { System.out.println(name + " says Woof!"); }
}
```

Using the Inherited Class

- The Dog object can access its own methods AND inherited methods seamlessly

```
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.name = "Tommy";
        d.eat();   // inherited from Animal
        d.bark();  // defined in Dog
    }
}
```

Output

```
Tommy is eating
Tommy says Woof!
```

What Gets Inherited?

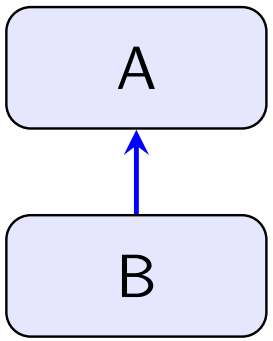
- Only accessible members are directly inherited by the subclass
- `private` members exist in memory but are NOT directly accessible
- Constructors are NOT inherited (subclass must define its own)

Access Modifier / Member	Condition	Inherited?
`public` / `protected`	Anywhere	**Yes**
`default` (package-private)	Subclass is in the same package	**Yes**
`private`	Exists in object, but not directly accessible	**No**
Constructors	Subclass must explicitly/implicitly call `super()`	**No**

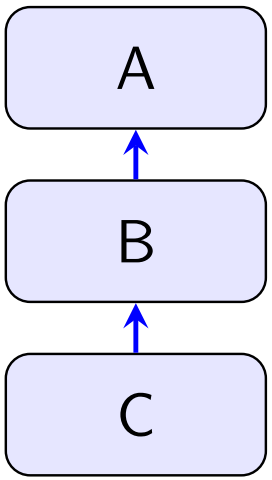
Types of Inheritance in Java: Overview

- **Single**: one subclass extends exactly one superclass
- **Multilevel**: a chain, e.g. Animal $\rightarrow$ Dog $\rightarrow$ Puppy
- **Hierarchical**: multiple subclasses extend the same single superclass
- *Note:* Full syntax and examples covered in the next lecture

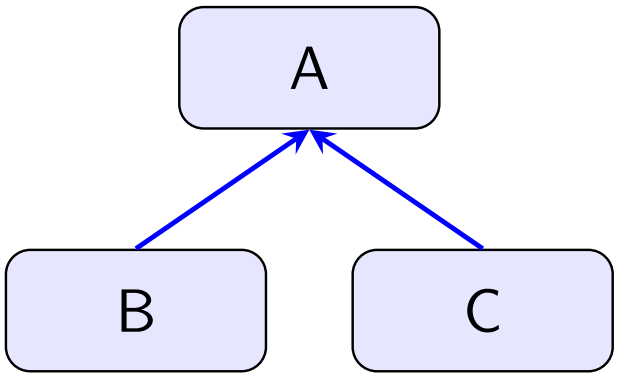
Single



Multilevel

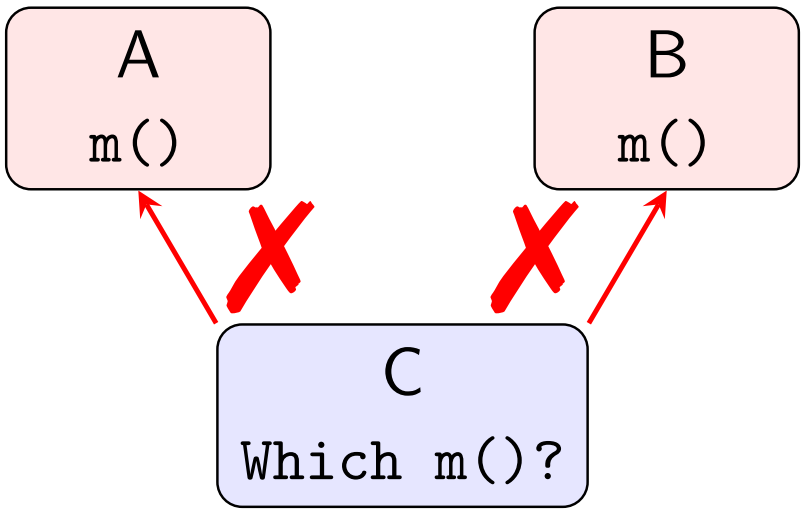


Hierarchical



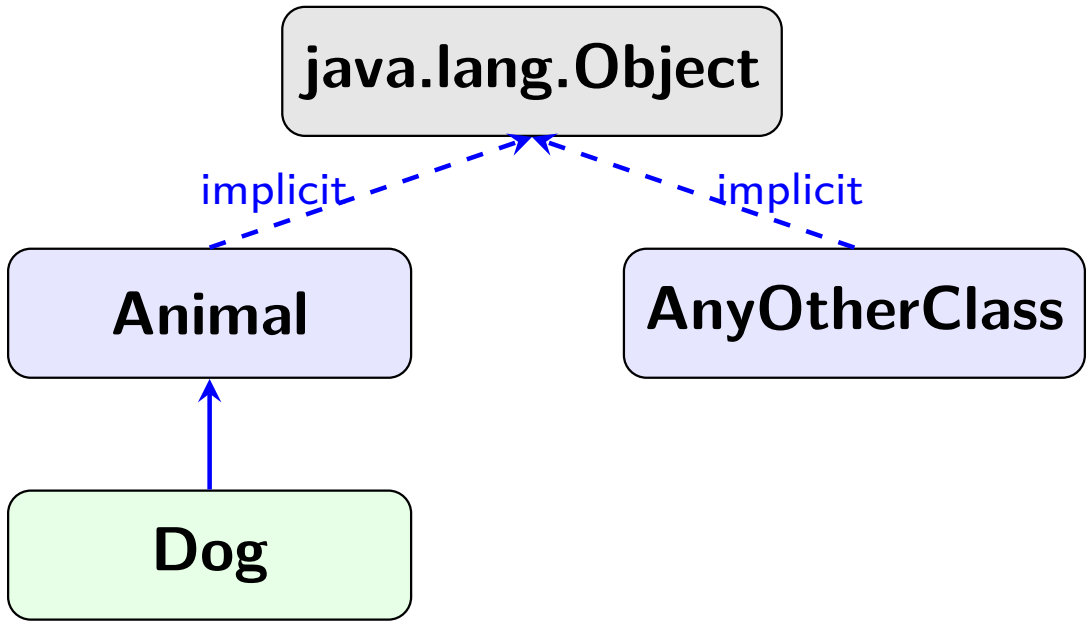
Java Does NOT Support Multiple Class Inheritance

- A Java class cannot extend more than one class (`class C extends A, B` is a compile error)
- **Reason**: the "diamond problem" - if A and B both define method `m()`, which version should C inherit?
- Java avoids ambiguity. Safe multiple-inheritance-like behaviour is achieved via **interfaces**



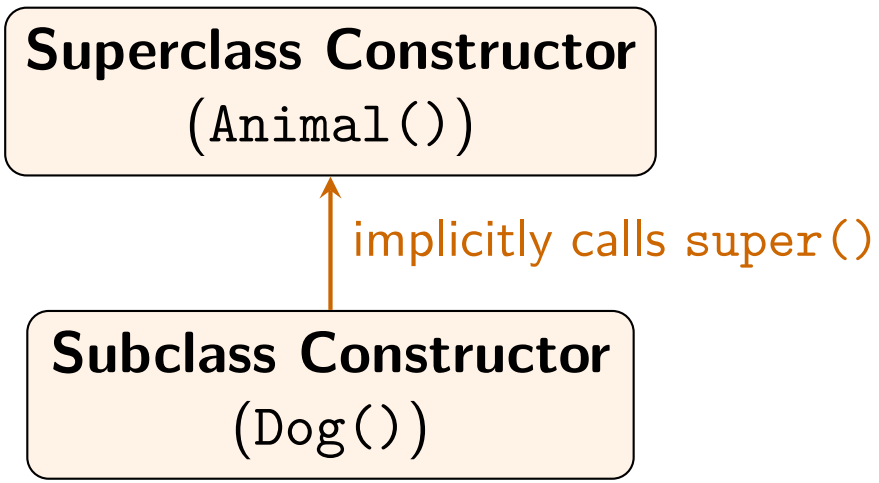
Every Class Inherits from Object

- If a class doesn't explicitly extend anything, it implicitly extends `java.lang.Object`
- `class Animal {}` is really `class Animal extends Object {}` behind the scenes
- Automatically gives every Java object methods like `toString()`, `equals()`, and `hashCode()`



Constructors and Field Shadowing: A Preview Warning

- Subclass constructor implicitly calls the superclass's no-argument constructor first
- If the superclass has no no-argument constructor, subclass MUST use `super(...)`
- **Field shadowing**: subclass declaring a field with the same name as a superclass field (causes confusing bugs, avoid it)



Common Mistakes and Exam Tips

- **Mistake**: writing `class Dog extends Animal, Pet` - Java allows only one class after `extends`
- **Mistake**: assuming `private` superclass fields can be accessed directly by name in the subclass
- **Exam tip**: always be ready to identify superclass vs subclass and justify the IS-A relationship in a given code snippet
- **Exam tip**: "Java supports single, multilevel, and hierarchical inheritance for classes, but not multiple inheritance of classes" is a common short-answer line

Key Takeaways

- Inheritance lets a subclass reuse a superclass's fields and methods, expressed through the **IS-A** relationship
- The `extends` keyword creates the inheritance relationship; `public`/`protected` members are inherited, `private` members and constructors are not directly
- Java supports single, multilevel, and hierarchical inheritance of classes, but never multiple inheritance of classes, to avoid the diamond problem
- Every class implicitly extends `java.lang.Object`, inheriting methods like `toString()`, `equals()`, and `hashCode()`

Next Lecture Preview

Lecture 20: Constructors in Inheritance & `super`

- Completes topic 3.1 by covering the `super` keyword
- Constructor chaining between superclass and subclass
- Worked examples of single, multilevel, and hierarchical inheritance

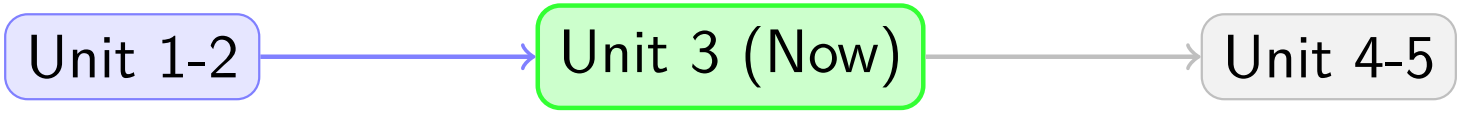
Questions?

Recap: extends and the IS-A Relationship

- Inheritance lets a subclass reuse a superclass's public/protected fields and methods via `extends`
- Private members and constructors of the superclass are not directly inherited
- Java supports single, multilevel, and hierarchical inheritance, but never multiple inheritance of classes
- Every class implicitly extends `java.lang.Object`

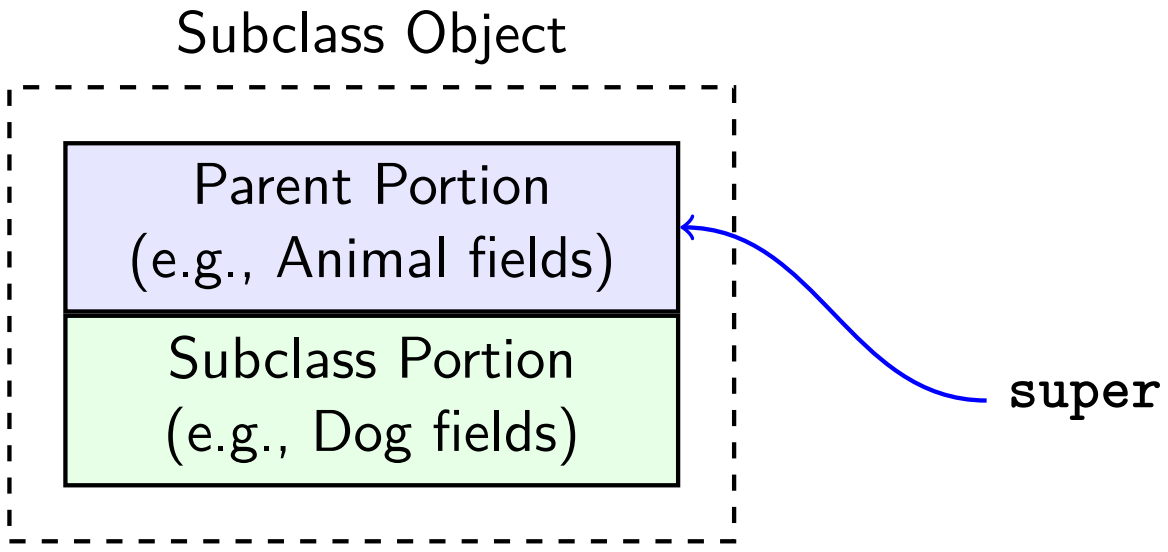
Course Overview & Today's Agenda

- Part 1: the `super` keyword - accessing parent fields/methods and `super()` constructor calls
- Constructor chaining: the order in which constructors execute in an inheritance chain
- Part 2: worked examples of single, multilevel, and hierarchical inheritance
- Comparison table of the three inheritance types



The 'super' Keyword: Accessing Parent Members

- `super` is a reference used inside a subclass to refer to its immediate superclass
- `super.fieldName` accesses a superclass field, useful when the subclass has a field with the same name
- `super.methodName(...)` calls the superclass's version of a method, even if overridden



super.method() in Action

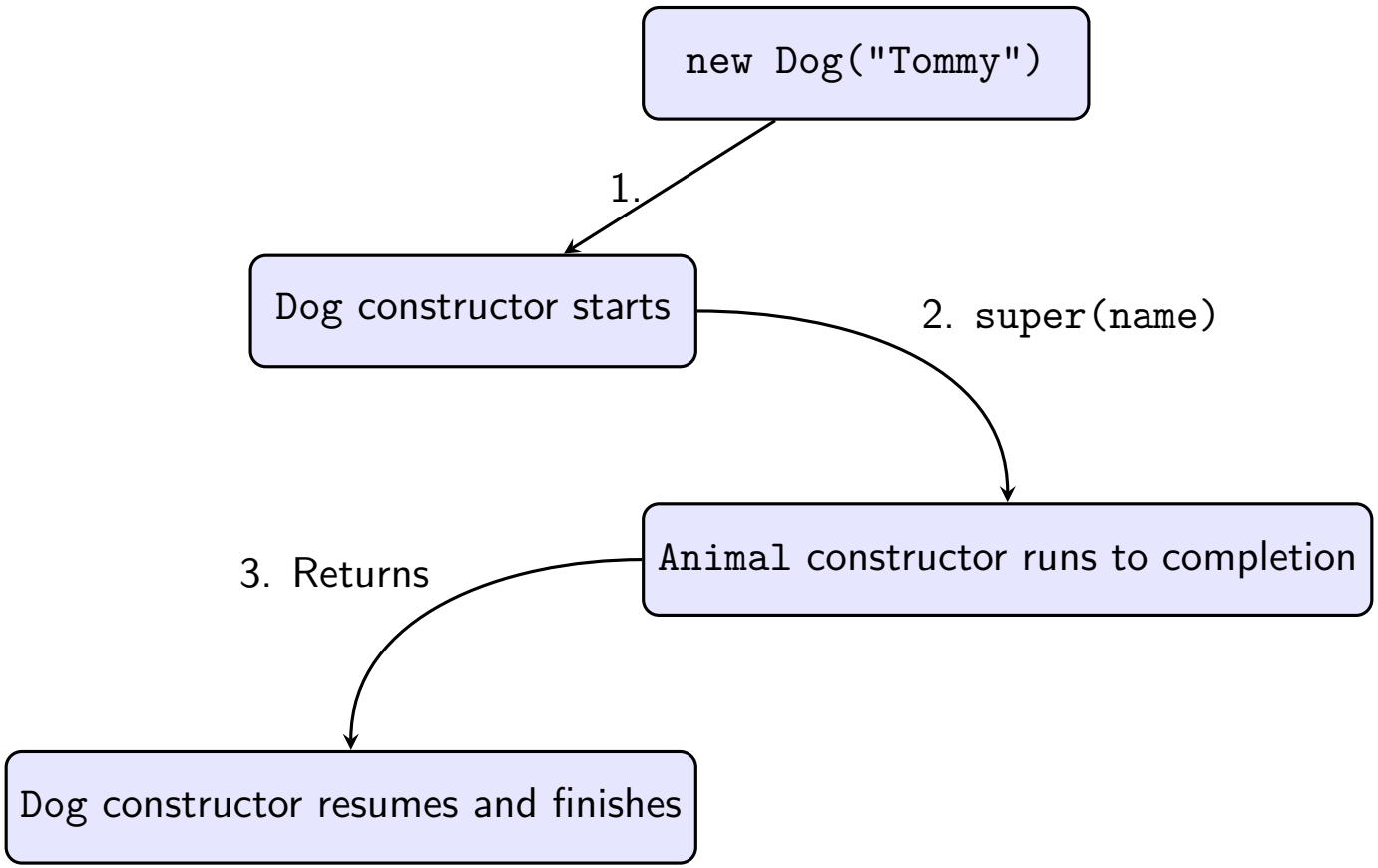
```
class Animal {
    void makeSound() { System.out.println("Some generic sound");
        }
}

class Dog extends Animal {
    void makeSound() {
        super.makeSound();              // calls Animal's version
            first
        System.out.println("Woof!");
    }
}
```

super() to Call the Superclass Constructor

```
class Animal {
    String name;
    Animal(String name) {
        this.name = name;           // recall Lecture 14's 'this'
            keyword
        System.out.println("Animal constructor");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);                // must be the FIRST
            statement
        System.out.println("Dog constructor");
    }
}
```

Constructor Chaining: Execution Order

- `new Dog("Tommy")` triggers this order:
- 1. `Dog` constructor begins
- 2. `super(name)` runs `Animal` constructor completely
- 3. Control returns to `Dog` constructor to finish



Implicit super() and the No-Argument Constructor Rule

- If a subclass constructor does not explicitly call `super(...)`, Java implicitly inserts a call to the superclass's no-argument constructor
- This fails to compile if the superclass has no no-argument constructor - the subclass must then explicitly call an available `super(args)` version
- This is why adding any parameterised constructor to a class can silently break existing subclasses

Rule of Thumb

If a superclass defines any constructor, consider whether a no-argument one is still needed for subclasses.

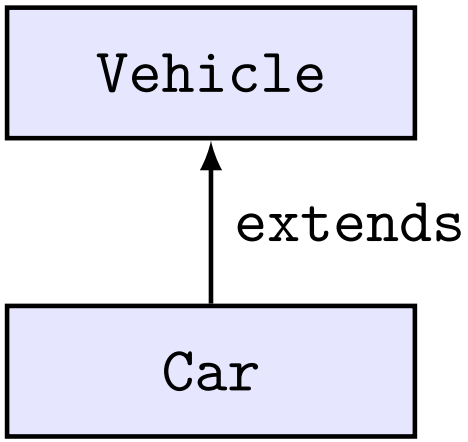
'this' vs 'super': Side-by-Side

`this`	`super`
Refers to the current object	Refers to the current object's parent portion
`this(...)` calls another constructor in the SAME class	`super(...)` calls a constructor in the PARENT class
`this.field` disambiguates a field from a parameter	`super.field` reaches a parent field hidden by subclass

Both must be the **FIRST** statement in a constructor (if used), and only one is allowed.

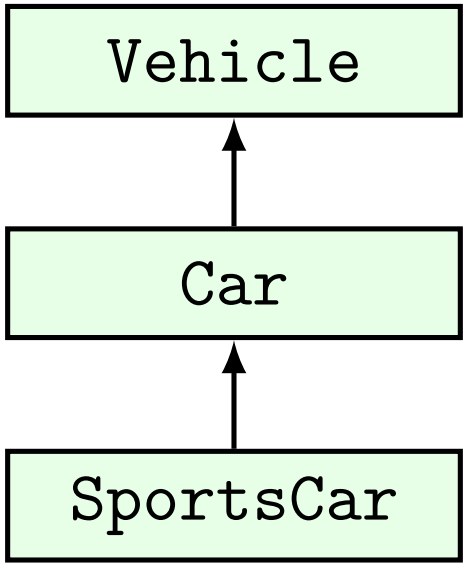
Type 1: Single Inheritance

```
class Vehicle {
    void start() { System.out.println("
        Vehicle starting"); }
}

class Car extends Vehicle {
    void drive() { System.out.println("
        Car driving"); }
}
// One subclass, one superclass -> single
     inheritance
```



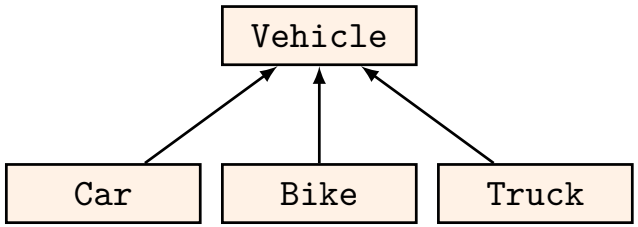
Type 2: Multilevel Inheritance

```
class Vehicle {
    void start() { System.out.println("
        Vehicle starting"); }
}
class Car extends Vehicle {
    void drive() { System.out.println("Car
        driving"); }
}
class SportsCar extends Car {
    void turboBoost() { System.out.println("
        Turbo boost!"); }
}
// SportsCar inherits from Car, which
    inherits from Vehicle
```



Type 3: Hierarchical Inheritance

```
class Vehicle {
    void start() { System.out.println("
        Vehicle starting"); }
}
class Car extends Vehicle { }
class Bike extends Vehicle { }
class Truck extends Vehicle { }
// Three subclasses share the SAME
    superclass Vehicle
```



Comparing the Three Inheritance Types

Type	Structure	Example
Single	One superclass, one subclass	`Vehicle` → `Car`
Multilevel	A chain of superclass/subclass levels	`Vehicle` → `Car` → `SportsCar`
Hierarchical	One superclass, many subclasses	`Vehicle` → `Car, Bike, Truck`

Common Mistakes and Exam Tips

- ✗ **Mistake:** placing statements before `super(...)` or `this(...)` in a constructor - both must be the first statement
- ✗ **Mistake:** confusing multilevel (a chain) with hierarchical (a fan-out) - draw the diagram to check
- ✓ **Exam tip:** be ready to trace constructor execution order for a 2-3 level inheritance chain and state the printed output
- ✓ **Exam tip:** remember Java has no multiple inheritance of classes (Lecture 19) - hierarchical inheritance is NOT the same thing and is fully legal

Key Takeaways

- The `super` keyword accesses a superclass's fields/methods (`super.x`) or invokes its constructor (`super(...)`), and `super(...)` must be the first statement in a subclass constructor
- Constructor chaining always runs the superclass constructor to completion before the subclass constructor body continues
- Java supports single (one-to-one), multilevel (chained), and hierarchical (one-to-many) inheritance of classes
- `this` refers to the current object and same-class constructors; `super` refers to the parent object and parent-class constructors - never confuse the two

Next Lecture Preview

- Lecture 21 opens topic 3.2 with method overriding - the rules for redefining a superclass method in a subclass - and dynamic method dispatch, the mechanism behind runtime polymorphism.

See you next time!

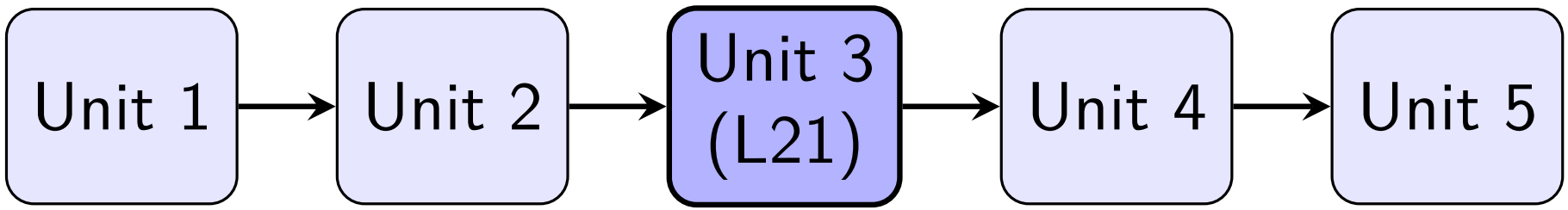
Questions?

Recap: super, Constructor Chaining, Inheritance Types

- `super` accesses parent fields/methods; `super(...)` calls parent constructor (must be first statement)
- Constructor chaining completes superclass constructor before subclass continues
- Java supports single, multilevel, and hierarchical inheritance

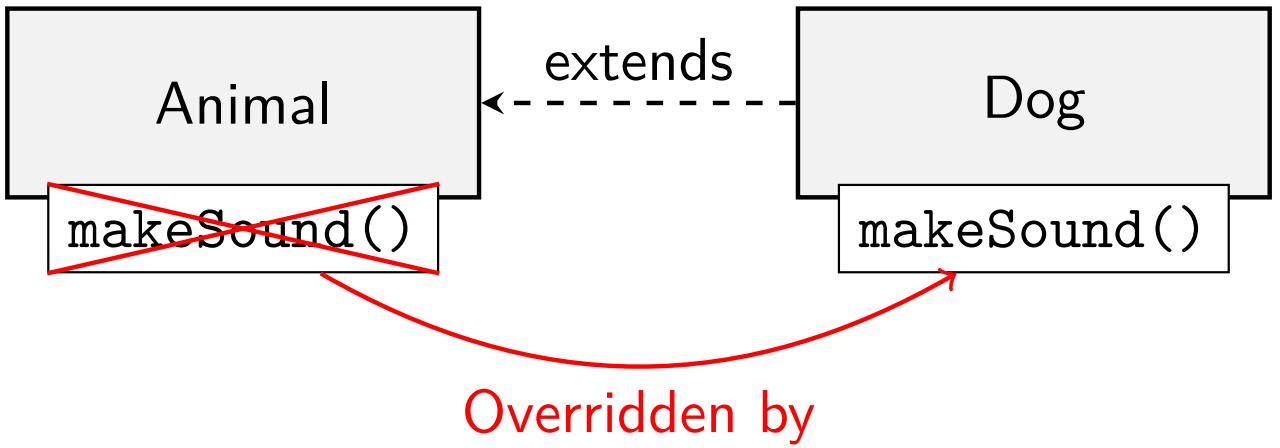
Course Overview & Today's Agenda

- Method overriding: definition and exact rules
- The `@Override` annotation and its importance
- Overriding vs overloading comparison
- Dynamic method dispatch: runtime method resolution



What Is Method Overriding?

- Subclass provides its own implementation of a superclass method
- Same name, same parameter list, compatible return type
- Customises inherited behaviour instead of adding new behaviour
- Achieves runtime (dynamic) polymorphism (OOP pillar)



Rules for a Valid Override

Override Checklist

- **Signature:** Name and parameter list (number, type, order) must exactly match
- **Return Type:** Same type or a subtype (covariant return type)
- **Access:** Cannot be MORE restrictive (e.g. cannot override public with private)
- **Exceptions:** Cannot throw broader checked exceptions

@Override Annotation

- Optional but recommended: compiler verifies override is valid
- Prevents silent errors (e.g., typos like `makeSond()`)

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }
}
```

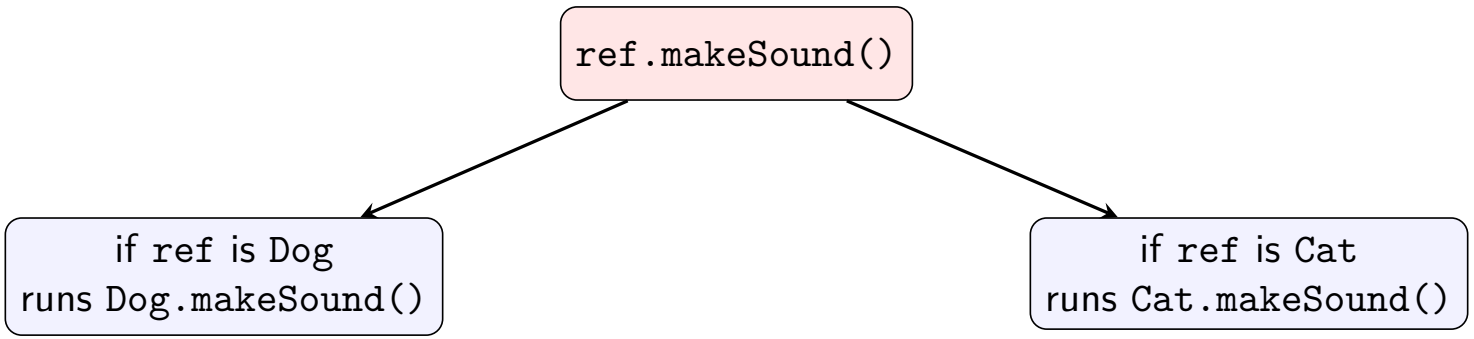
Overloading vs Overriding: Side-by-Side

Feature	Overloading (Lecture 17)	Overriding (Today)
Location	Same class	Superclass/Subclass pair
Method Name	Same	Same
Parameters	Different	Exactly the Same
Resolution	Compile time	Run time
Polymorphism	Compile-time polymorphism	Runtime polymorphism

- Overloading requires signature differences; Overriding requires exact signature match

Dynamic Method Dispatch: The Idea

- Java decides at **RUN time** which overridden method to execute
- Decision based on **actual object type**, NOT reference variable's type
- Also called "runtime polymorphism" or "late binding"
- A single line of code behaves differently based on referenced object



Dynamic Dispatch in Action

```
class Animal {
    void makeSound() { System.out.println("Some generic sound");
        }
}
class Dog extends Animal {
    @Override
    void makeSound() { System.out.println("Woof!"); }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();    // reference Animal, object Dog
        a.makeSound();           // prints 'Woof!' - Dog version
            runs
    }
}
```

Calling the Parent Version with super

- Use `super.methodName()` to extend (not just replace) behaviour

```
class Dog extends Animal {
    @Override
    void makeSound() {
        super.makeSound();    // explicitly run Animal's version
            too
        System.out.println("Woof!");
    }
}
// Output: 'Some generic sound' followed by 'Woof!'
```

Methods That Cannot Be Overridden

Non-overridable methods

- **`static` methods:** Belong to class, not object. Hidden by subclass version, not overridden (dispatch is by reference type).
- **`final` methods:** Explicitly locked against overriding (Lecture 18).
- **`private` methods:** Not inherited (Lecture 19); same-named method in subclass is entirely separate.
- **Constructors:** Never overridden; each class defines its own, chained via `super()` (Lecture 20).

Common Mistakes and Exam Tips

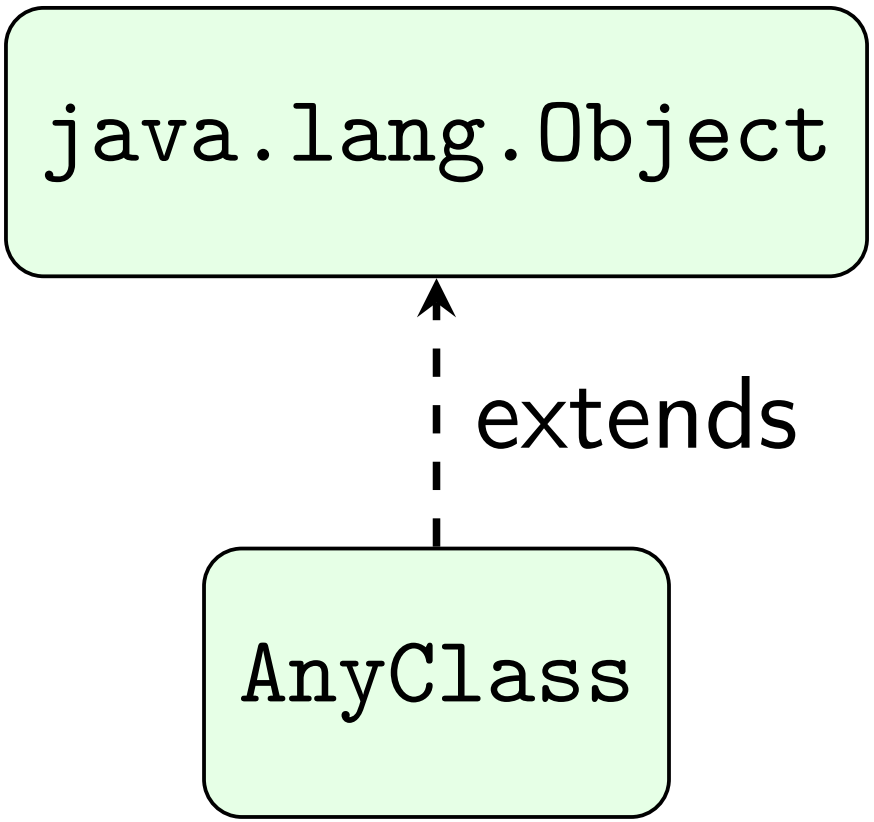
- **Mistake:** Changing parameter list slightly → that's overloading (new method)
- **Mistake:** Forgetting `@Override` → no warning for typos
- **Exam tip:** For `Animal a = new Dog(); a.makeSound();`, always trace the **actual OBJECT type**, not reference type, to find output
- **Exam tip:** Know the 3 reasons a method might fail to override (`static`, `final`, `private`) and how it differs from overloading

Key Takeaways

- Overriding needs same name, same parameters, compatible return type; use `@Override` to verify
- Overriding (runtime, same signature) $\neq$ Overloading (compile-time, different signature)
- Dynamic dispatch: Java chooses method based on object type at runtime, not reference type
- `static`, `final`, `private` methods cannot be overridden

Next Lecture Preview

- Lecture 22 completes topic 3.2
- Examines the `Object` class that every Java class implicitly extends
- Shows how to properly override `toString()`, `equals()`, and `hashCode()`



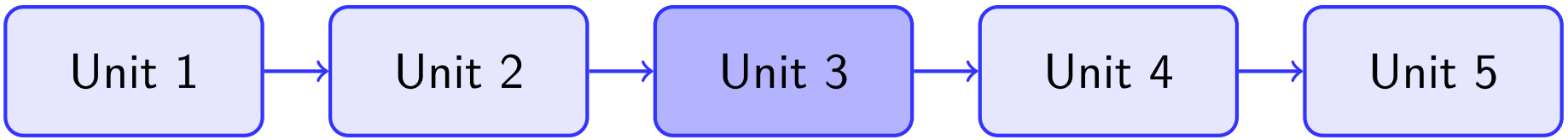
Questions?

Recap: Overriding and Dynamic Dispatch

- Overriding requires matching name, parameters, and compatible return type; @Override lets the compiler verify it
- Overriding (runtime) is different from overloading (compile-time, Lecture 17)
- Dynamic method dispatch: Java runs the overridden version matching the object's actual runtime type

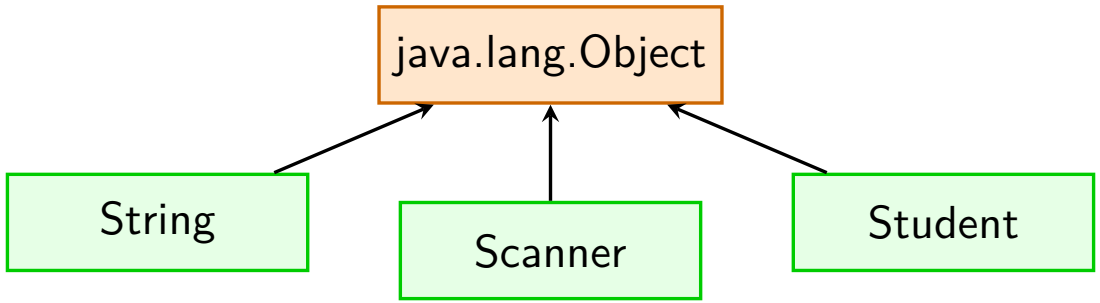
Course Overview & Today's Agenda

- The Object class as the implicit root of every Java class (recall Lecture 19)
- Overriding toString() for readable object output
- Overriding equals() for content-based comparison
- The equals()/hashCode() contract and why both matter together



java.lang.Object: The Root of Everything

- Every Java class implicitly extends Object if not explicitly extending anything else
- Every object automatically inherits a set of default methods from Object
- Key inherited methods: toString(), equals(Object obj), hashCode(), getClass()
- These defaults are usually not useful as-is - Java lets us override them



Default toString(): Not Very Useful

```
class Student {
    String name; int roll;
    Student(String name, int roll) { this.name = name; this.roll
        = roll; }
}

Student s = new Student("Asha", 12);
System.out.println(s);
// Default output: Student@1b6d3586 (class name @ hashcode in
    hex)
```

Overriding toString()

```
class Student {
    String name; int roll;
    Student(String name, int roll) { this.name = name; this.roll
        = roll; }

    @Override
    public String toString() {
        return "Student[name=" + name + ", roll=" + roll + "]";
    }
}
// System.out.println(s) now prints: Student[name=Asha, roll=12]
```

Default equals(): Reference Comparison

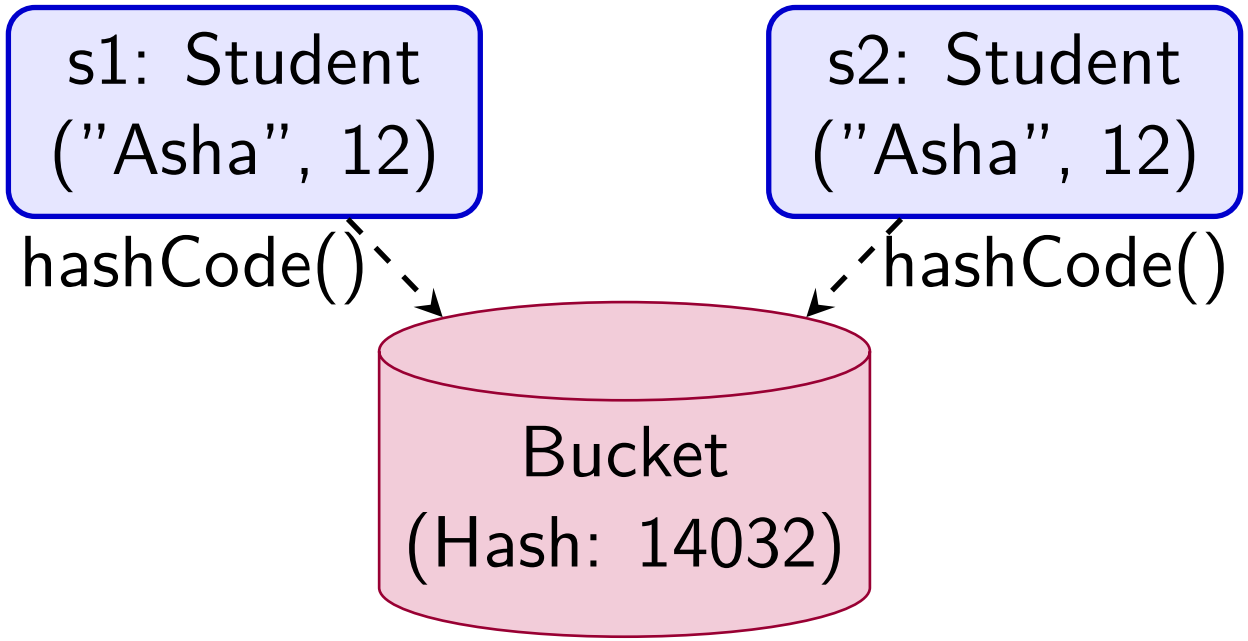
```
Student s1 = new Student("Asha", 12);
Student s2 = new Student("Asha", 12);

System.out.println(s1 == s2);          // false - different
    objects in memory
System.out.println(s1.equals(s2));     // false by default too!
// Object's default equals() just checks if it's the SAME object
     (like ==)
```

Overriding equals()

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return
        false;
    Student other = (Student) obj;
    return roll == other.roll && name.equals(other.name);
}
// Now s1.equals(s2) returns true - compares by content
```

hashCode(): The Object's 'Fingerprint'

- `hashCode()` returns an int that acts as a numeric fingerprint
- Default `hashCode()` is typically derived from the memory address
- **Contract:** if two objects are equal via `equals()`, they MUST return the same `hashCode()`
- Unequal objects MAY share a hashCode (collision)



Overriding hashCode() to Match equals()

```
@Override
public int hashCode() {
    return Objects.hash(name, roll);
}
// java.util.Objects.hash(...) combines multiple fields into one
     hashCode
// Must use the SAME fields here as compared in equals()
```

Why Overriding Both Together Matters

- If equals() is overridden but hashCode() is not, two 'equal' objects can have different hashCodes
- This breaks the contract and causes subtle bugs in hash-based collections
- Modern IDEs and Java records can auto-generate correct pairs

Exam Rule to Remember

Always override equals() and hashCode() together, never just one. Equal objects MUST produce equal hashCodes.

getClass() and instanceof: Related Object Tools

- `getClass()` returns the runtime Class object - used to safely guard equals()
- The `instanceof` operator checks if an object is of a given type before casting - avoids ClassCastException
- Both are commonly used with equals() overrides and dynamic dispatch patterns

```
// Example pattern:
if (obj instanceof Student) {
    Student s = (Student) obj;
    // safe to use s.roll and s.name here
}
```

Common Mistakes and Exam Tips

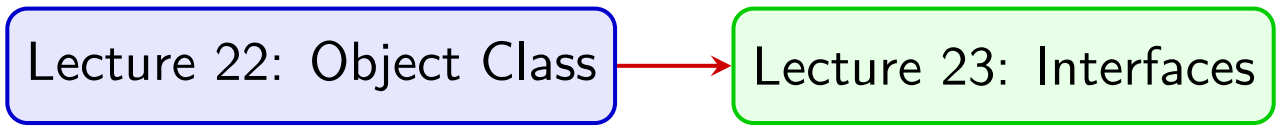
- **Mistake:** overriding equals() but forgetting hashCode(), breaking the equals/hashCode contract
- **Mistake:** comparing objects with == when content equality (equals()) was actually intended
- **Exam tip:** be ready to write a complete toString() or equals() override for a simple class given its fields
- **Exam tip:** state the equals/hashCode contract precisely - "equal objects must have equal hashcodes"

Key Takeaways

- java.lang.Object is the implicit root, providing default toString(), equals(), and hashCode() methods
- Overriding toString() gives objects a meaningful printed representation instead of ClassName@hashcode
- Overriding equals() enables content-based comparison instead of default reference (==) comparison
- The equals()/hashCode() contract requires that equal objects produce equal hashCodes

Next Lecture Preview

- Lecture 23 opens topic 3.3 by introducing interfaces
- Interfaces let a Java class achieve multiple-inheritance-like behaviour safely
- This resolves the restriction of multiple class inheritance ruled out in Lecture 19



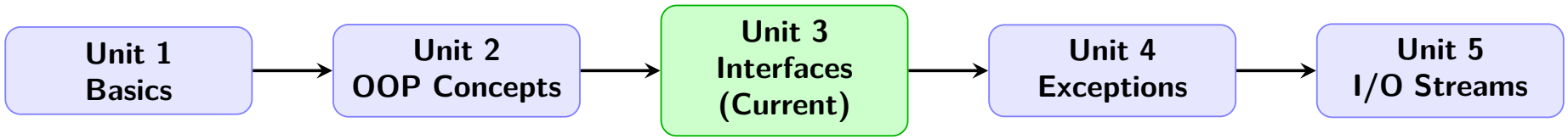
Questions?

Recap: Object Class and Its Methods

- Every class implicitly extends `Object`, inheriting `toString()`, `equals()`, `hashCode()`
- Override `toString()` for readable output; override `equals()` for content-based comparison
- `equals()` and `hashCode()` must always be overridden together to satisfy their contract

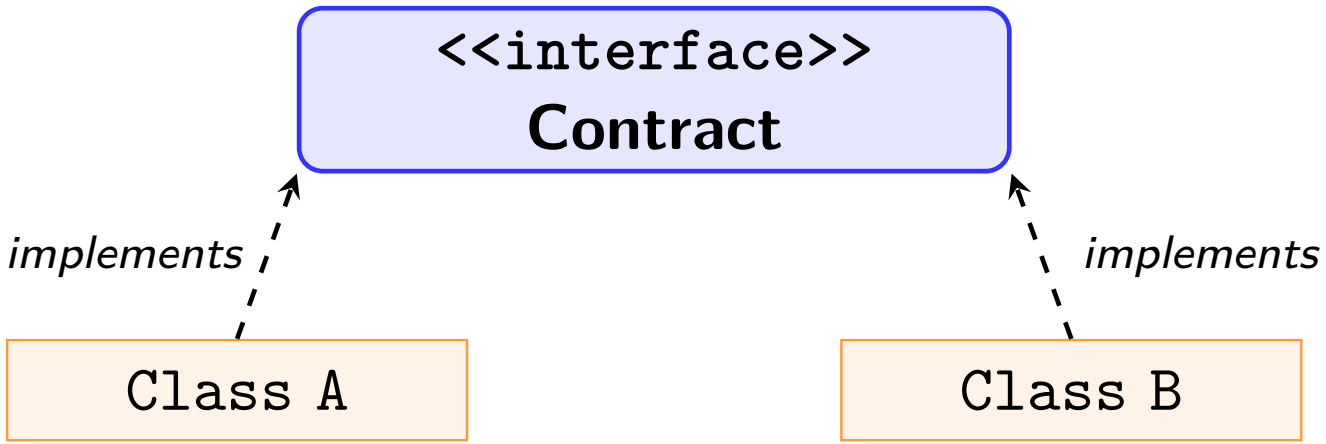
Course Overview & Today's Agenda

- What is an interface, and how is it different from a class?
- Interface syntax: defining an interface and implementing it with `implements`
- Implementing multiple interfaces in one class
- How interfaces solve the multiple-inheritance problem



What Is an Interface?

- Abstract type defining **WHAT** a class must do, not **HOW**
- Pre-Java 8: contains only abstract method signatures and constants
- Implementing = signing a **contract** to provide real code for methods
- Models **capability** (e.g., `Flyable`) rather than strict IS-A relationship



Interface Syntax: Defining an Interface

Interface Basics

Use the keyword `interface` instead of `class`. Methods have no body.

```
interface Flyable {
    void fly();             // implicitly public and abstract
}
```

Implementing an Interface with 'implements'

```
class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird flaps wings and flies");
    }
}

Flyable f = new Bird();
f.fly();   // 'Bird flaps wings and flies'
```

Rules Inside an Interface

- Methods are implicitly **public** and **abstract** (no body) (unless default/static)
- Fields are implicitly **public, static, and final** (named constants)
- Cannot be instantiated directly: `new Flyable()` is a compile error
- Implementing class must provide a **public** implementation for every abstract method, or be declared abstract itself

Implementing Multiple Interfaces

```
interface Flyable { void fly(); }
interface Swimmable { void swim(); }

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() { System.out.println("Duck flies short
        distances"); }
    @Override
    public void swim() { System.out.println("Duck swims
        gracefully"); }
}
```

Interface vs Class: Syntax Differences

Class	Interface
Uses keyword `class`	Uses keyword `interface`
Can hold state (instance fields) and full method bodies	(Pre-Java 8) No instance fields or method bodies
Supports single `extends` only (at most one superclass)	A class can `implements` MANY interfaces

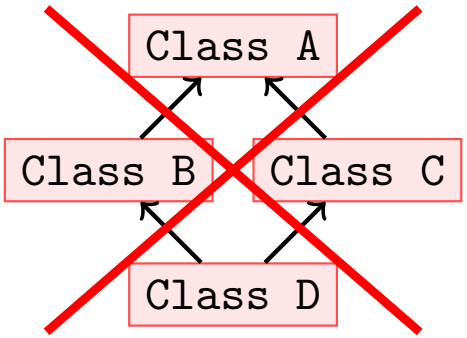
Complementary Mechanisms

A class can BOTH `extend` one class AND `implement` multiple interfaces at the same time.

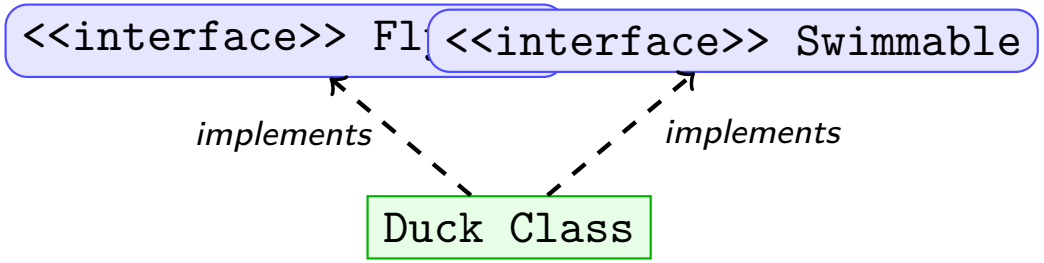
Multiple Inheritance Through Interfaces: Why It's Safe

- **Problem:** Diamond problem with classes (ambiguous inherited method **bodies**).
- **Solution:** Interfaces have no method bodies. No ambiguous **code** to inherit!
- A class simply promises to write methods itself—no conflict possible.

Diamond Problem (Ambiguous)



Safe (No Body Conflicts)



Using instanceof with Interfaces

```
Duck d = new Duck();
System.out.println(d instanceof Flyable);    // true
System.out.println(d instanceof Swimmable);  // true
// A single object can satisfy multiple interface types at once
```

Common Mistakes and Exam Tips

- **Mistake:** forgetting to mark implementing method `public` (interface methods are implicitly public and cannot be narrowed)
- **Mistake:** trying to instantiate an interface directly with `new InterfaceName()`
- **Exam tip:** be ready to write a two-interface example (like `Flyable`/`Swimmable`) and explain why it avoids the diamond problem
- **Exam tip:** "Interfaces achieve 100% abstraction and support multiple inheritance of type" is a standard short-answer line

Key Takeaways

- An interface is a contract of implicitly public abstract methods (and public static final constants) that a class fulfils using `implements`
- A class can implement multiple interfaces at once, even though it can extend only one class
- Interfaces avoid the diamond problem because (pre-Java 8) they carry no method bodies, so there is nothing ambiguous to inherit
- A class can both extend one superclass and implement several interfaces simultaneously

Next Lecture Preview

Default and Static Methods in Interfaces

- Lecture 24 completes topic 3.3
- Learn how Java 8 added `default` and `static` methods to let interfaces carry actual method bodies
- Discover the rules for resolving conflicts when two interfaces provide the same default method

Questions?

Recap: Interfaces as Contracts

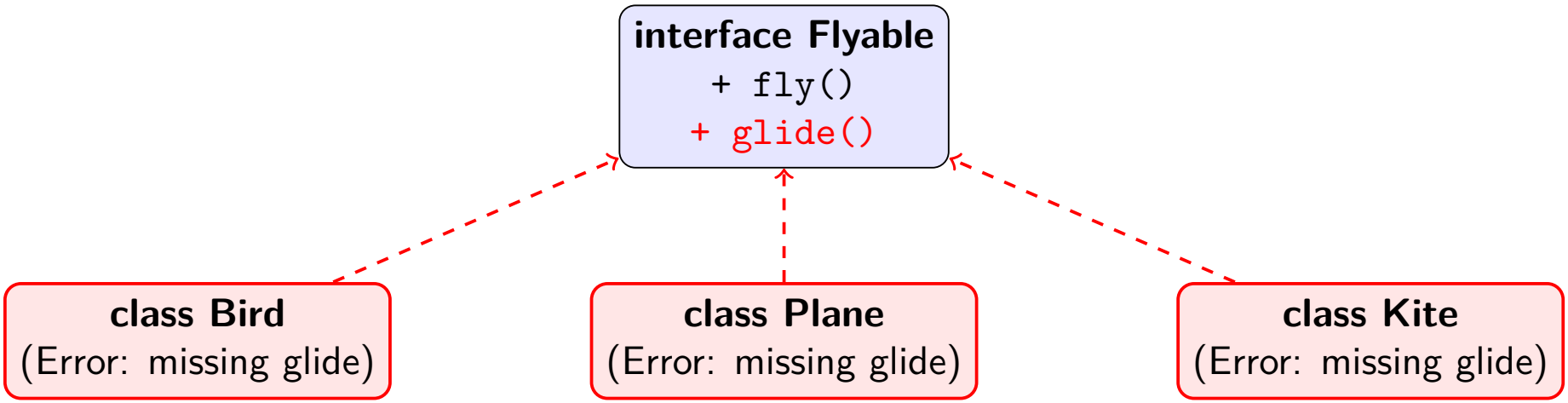
- An interface defines abstract method signatures that implementing classes must fulfil via `implements`
- A class can implement multiple interfaces, unlike single class inheritance
- Pre-Java-8 interfaces had no method bodies, avoiding the diamond problem entirely

Course Overview & Today's Agenda

- The backward-compatibility problem motivating default methods
- `default` method syntax and worked examples
- `static` method syntax inside interfaces
- Resolving conflicts between clashing default methods

The Problem: Adding a Method to an Existing Interface

- Suppose `Flyable` is used by hundreds of existing classes
- Adding abstract `glide()` breaks EVERY implementing class
- Java 8 solved this backward-compatibility problem with default methods



default Methods: Syntax

- default methods allow interfaces to provide a body

```
interface Flyable {
    void fly();                              // still abstract, must be
        implemented

    default void glide() {
        System.out.println("Gliding using default behaviour");
    }
}
```

Using a default Method

- Classes inherit default methods for free, ensuring backward compatibility

```
class Bird implements Flyable {
    @Override
    public void fly() { System.out.println("Bird flies"); }
    // glide() is NOT re-implemented - Bird uses Flyable's
        default for free
}

Bird b = new Bird();
b.fly();      // 'Bird flies'
b.glide();    // 'Gliding using default behaviour' - inherited
    default
```

static Methods in Interfaces

- Interfaces can also have `static` methods with implementations

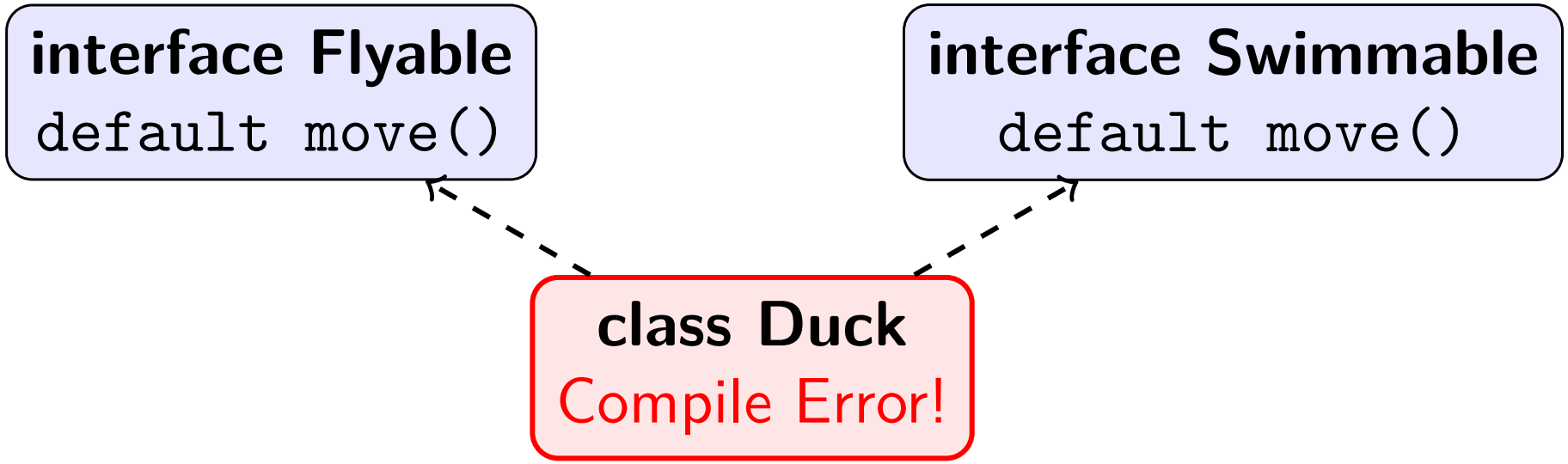
```
interface MathUtil {
    static int square(int x) {
        return x * x;
    }
}

int result = MathUtil.square(5);    // called on the interface
    name, like Math.sqrt()
```

static Interface Methods: Key Rules

- A `static` method in an interface must have a body (fully implemented)
- Called using the interface name directly (`InterfaceName.method()`), never through an implementing object
- NOT inherited by implementing classes - it belongs only to the interface itself
- Typically used for utility/helper methods related to the interface

The Diamond Problem Returns... Almost

- Two interfaces could define a `default` method `move()` with different bodies
- If a class implements BOTH, Java refuses to silently guess, causing a COMPILE ERROR
- Java forces explicit resolution, unlike C++-style multiple inheritance



Resolving a Default Method Conflict

- Must override the clashing method, explicitly choosing a parent's version

```
interface Flyable { default void move() { System.out.println("
    Flying"); } }
interface Swimmable { default void move() { System.out.println("
    Swimming"); } }

class Duck implements Flyable, Swimmable {
    @Override
    public void move() {
        Flyable.super.move();   // explicitly choose Flyable's
            version
        System.out.println("Duck decides how to move");
    }
}
```

Interface Method Kinds: Summary Table

Kind	Has body?	Called via	Inherited by class?
abstract (plain)	No	implementing object	Must be implemented
default	Yes	implementing object	Yes, unless overridden
static	Yes	interface name only	No

Common Mistakes and Exam Tips

Keep in mind

- **Mistake**: Trying to call a static interface method through an object reference instead of the interface name
- **Mistake**: Implementing two interfaces with a clashing default method and not overriding it - causes a compile error

- **Exam tip**: `default` methods enable backward-compatible interface evolution
- **Exam tip**: Know the `InterfaceName.super.methodName()` syntax for explicitly choosing a specific interface's default implementation

Key Takeaways

- **default methods**, introduced in Java 8, let interfaces provide a method body so new methods can be added without breaking existing classes
- **static methods** in interfaces are fully implemented utility methods called via the interface name and are never inherited
- A class implementing two interfaces with a clashing default method must explicitly override it, optionally using `InterfaceName.super.method()`
- Interfaces now support three method kinds: **abstract, default, and static**

Next Lecture Preview

- Lecture 25 moves to topic 3.4, introducing **abstract classes** and **abstract methods**
- Concrete methods within an abstract class
- A direct comparison of when to choose an abstract class versus an interface

Questions?

Recap: default and static Interface Methods

- **default methods**: give interfaces a method body, inherited by classes, enabling backward-compatible changes
- **static methods**: fully implemented utilities called via the interface name, never inherited
- Method clash between interfaces must be resolved with `InterfaceName.super.method()`

Course Overview & Today's Agenda

- The `abstract` keyword for classes and methods
- Concrete (fully implemented) methods living alongside abstract methods
- Why an abstract class cannot be instantiated, and what subclasses must do
- Abstract class vs interface: full side-by-side comparison and usage rules

What Is an Abstract Class?

- Declared with `abstract`; cannot be instantiated
- Mixes abstract methods (no body) with concrete methods (full body)
- Has fields and constructors (unlike interfaces) for shared logic

abstract class Shape

`String color;`

abstract double area(); (dashed/abstract)

`void describe();` (solid/concrete)

Declaring an Abstract Class and Method

```
abstract class Shape {
    String color;

    Shape(String color) { this.color = color; }   //
        constructors ARE allowed

    abstract double area();   // no body - must be implemented
        by subclasses
}
```

Concrete Methods Inside an Abstract Class

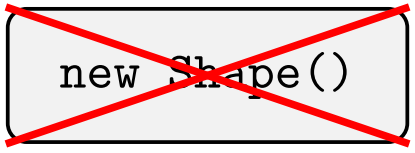
```
abstract class Shape {
    String color;
    Shape(String color) { this.color = color; }

    abstract double area();                    // abstract - no body

    void describe() {                           // CONCRETE - full
        body, shared by all subclasses
        System.out.println(color + " shape with area " + area())
            ;
    }
}
```

Implementing an Abstract Class

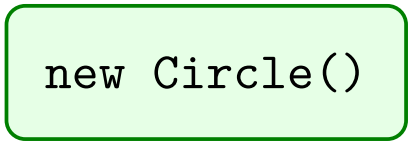
```
class Circle extends Shape {
    double radius;
    Circle(double radius) { super("Red"); this.radius = radius;
        }

    @Override
    double area() { return Math.PI * radius * radius; }
}

Shape s = new Circle(5);
s.describe();   // 'Red shape with area 78.5...' - uses
    inherited concrete method
```

Why Can't We Instantiate an Abstract Class?

- `new Shape("Blue")` is a compile error; abstract classes are incomplete
- Abstract methods like `area()` have no defined behaviour
- A subclass must implement ALL abstract methods to become instantiable
- A `Shape` reference variable is still perfectly legal and useful



Error!



Allowed

```
Shape s = new Circle();
```

Polymorphic Reference

Rules for Subclasses of an Abstract Class

- Concrete subclasses **MUST override** every abstract method
- A subclass MAY itself remain `abstract`, deferring implementation
- Concrete methods from the abstract class are inherited normally
- Abstract class constructors are used via `super()`, never directly

Multilevel Abstraction

`abstract Shape` → `abstract Polygon` → `concrete Triangle`

Abstract Class vs Interface: The Core Trade-off

- **Decision Rule:** Use abstract class for shared state/code; interface for common capability.

Abstract Class	**Interface**
Can have instance fields, constructors, any access modifier	(Mostly) no instance fields, no constructors
A class can `extend` only ONE abstract class	A class can `implement` MANY interfaces
Models a strong IS-A relationship with SHARED implementation	Models a capability/contract

Abstract Class vs Interface: Full Comparison Table

Feature	Abstract Class	Interface
Keyword	`abstract class`	`interface`
Multiple inheritance	Only one (`extends`)	Many (`implements`)
Fields	Any (instance, static, final)	`public static final` only
Constructors	Allowed	Not allowed
Method bodies	Concrete + abstract mixed	default/static (Java 8+) + abstract

Combining Abstract Classes and Interfaces

```
interface Drawable { void draw(); }

abstract class Shape implements Drawable {
    abstract double area();
    void describe() { System.out.println("Area: " + area()); }
}

class Circle extends Shape {
    double radius;
    Circle(double r) { radius = r; }
    double area() { return Math.PI * radius * radius; }
    public void draw() { System.out.println("Drawing a circle");
        }
}
```

Common Mistakes and Exam Tips

- **Mistake:** forgetting to mark a class `abstract` when it contains at least one abstract method (compile error)
- **Mistake:** trying to instantiate an abstract class directly
- **Exam Tip:** be ready to write a complete abstract class example and its concrete subclass
- **Exam Tip:** 'abstract class vs interface' is a guaranteed comparison question - know the tables!

Key Takeaways

- Abstract classes mix abstract methods (no body) with concrete methods
- Concrete subclasses must implement every inherited abstract method
- Abstract classes support single inheritance with shared state; interfaces support multiple implementation of a pure capability
- An abstract class can implement interfaces, and the two mechanisms combine freely

Next Lecture Preview

- **Lecture 26:** opens topic 3.5 by covering how to create and name Java packages, and how the `import` statement and package-level access control work together to organise larger codebases.

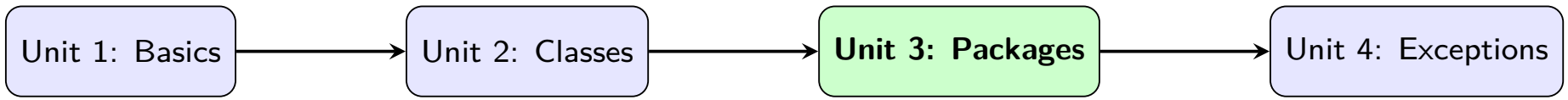
Questions?

Recap: Abstract Classes vs Interfaces

- Abstract classes mix abstract/concrete methods, support single inheritance
- Interfaces are pure contracts for multiple implementation
- Choose abstract classes for shared implementation, interfaces for capability

Course Overview & Today's Agenda

- Why packages exist: organisation, conflicts, access control
- Creating packages & naming conventions
- Directory structure, compilation, and wildcard imports



Why Do We Need Packages?

- A package is a namespace grouping related classes
- **Organisation**: logically related classes live together
- **Avoids conflicts**: different packages can use the same class name
- **Access control**: regulates visibility between class groups

Creating a Package: The 'package' Statement

```
package com.college.student;

public class Student {
    String name;
    public Student(String name) { this.name = name; }
}
```

- package statement must be the very first line
- Omitting it places code in the default (unnamed) package

Package Naming Conventions

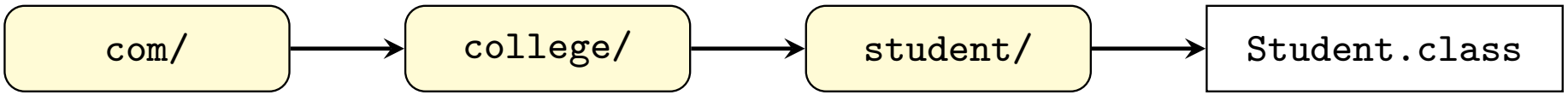
- **Reversed Domain:** Use reversed domain name as prefix (e.g., `com.college.student`)
- **Lowercase:** Always lowercase to avoid clashing with class names
- **No CamelCase:** Keep segments short (e.g., `studentmanagement`)
- **Prevention:** Stops collisions between packages from different teams

Example Real-world Packages

`java.util`, `java.io`, `com.google.gson`

Directory Structure Must Match the Package

- Package name maps directly to folder structure
- Example: `com.college.student` → `com/college/student/`
- The compiled `.class` file must sit inside that matching path



Compiling and Running Packaged Classes

```
javac -d . Student.java
// -d . tells javac to create the package folder structure
     starting from the current directory

java com.college.student.Student
// running a packaged class requires its FULLY QUALIFIED name
     from the base directory
```

- Use -d . to automatically create nested folders
- Run using the **fully qualified name** (package + class)

The 'import' Statement: Using Classes from Other Packages

```java
package com.college.admin;

import com.college.student.Student;   // single-class import

public class Admin {
    void register() {
        Student s = new Student("Asha");   // usable directly by
            simple name now
    }
}
```

- Imports follow the package statement
- Allows using a short name instead of fully qualified name

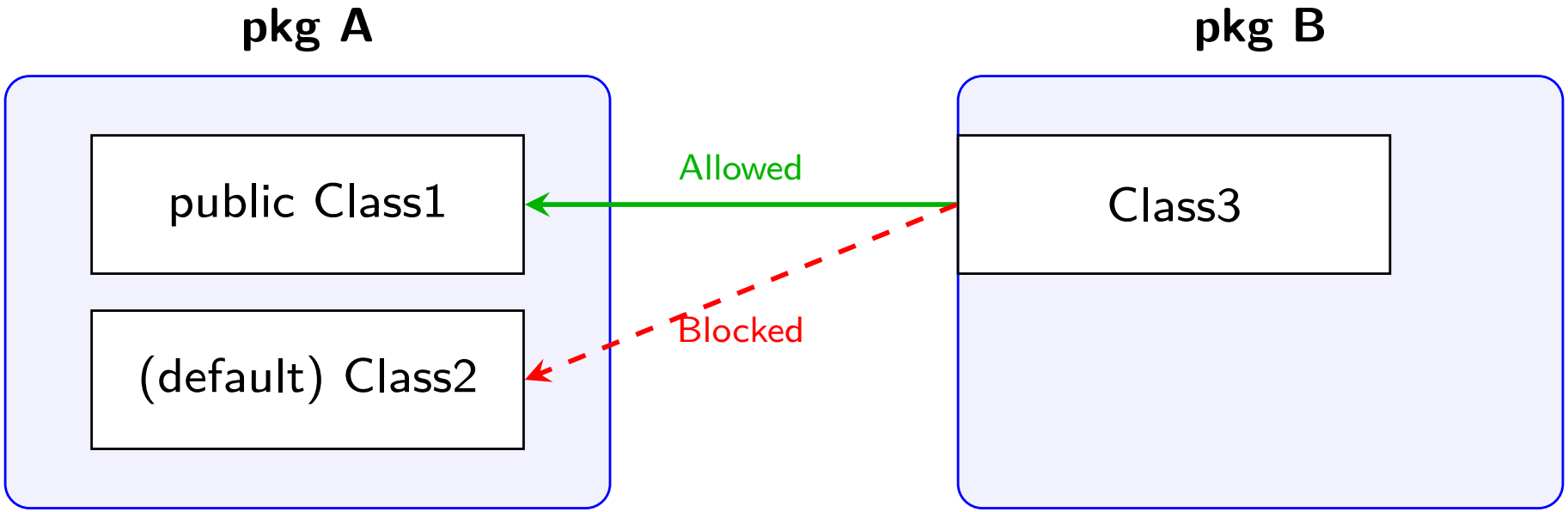
Wildcard Imports and Fully Qualified Names

```
import com.college.student.*;    // imports ALL public classes/
    interfaces in that package

// Alternative: skip import entirely and use the fully qualified
    name inline
com.college.student.Student s2 = new com.college.student.Student
    ("Ravi");
```

- **Wildcard Import (*):** Imports all classes within a package
- **Fully Qualified Name:** Can be used without an import
- Useful if two imported packages contain identically named classes

Package Access and the Default Modifier

- Default (package-private) is visible **only within the same package**
- Package structure directly controls visibility of these members
- Public members are visible everywhere; package-private are restricted



Common Mistakes and Exam Tips

- **Mistake:** Placing `package` anywhere except the first line
- **Mistake:** Mismatched folders & package name, causing class-not-found errors
- **Exam tip:** Know standard naming convention (lowercase, reversed domain)
- **Exam tip:** Explain default access tying directly to package structure

Key Takeaways

- Packages group related classes, prevent conflicts, and control visibility
- `package` statement must be first; folder must match name
- `import` statements allow using short class names
- Package-private access restricts visibility strictly to the same package

Next Lecture Preview

- Lecture 27 completes Unit 3 by touring Java's built-in packages: `java.lang`, `java.util`, `java.io`
- Unit wrap-up before beginning Exception Handling

Up Next: Built-in Packages!

Questions?

Recap: Creating Packages and Importing

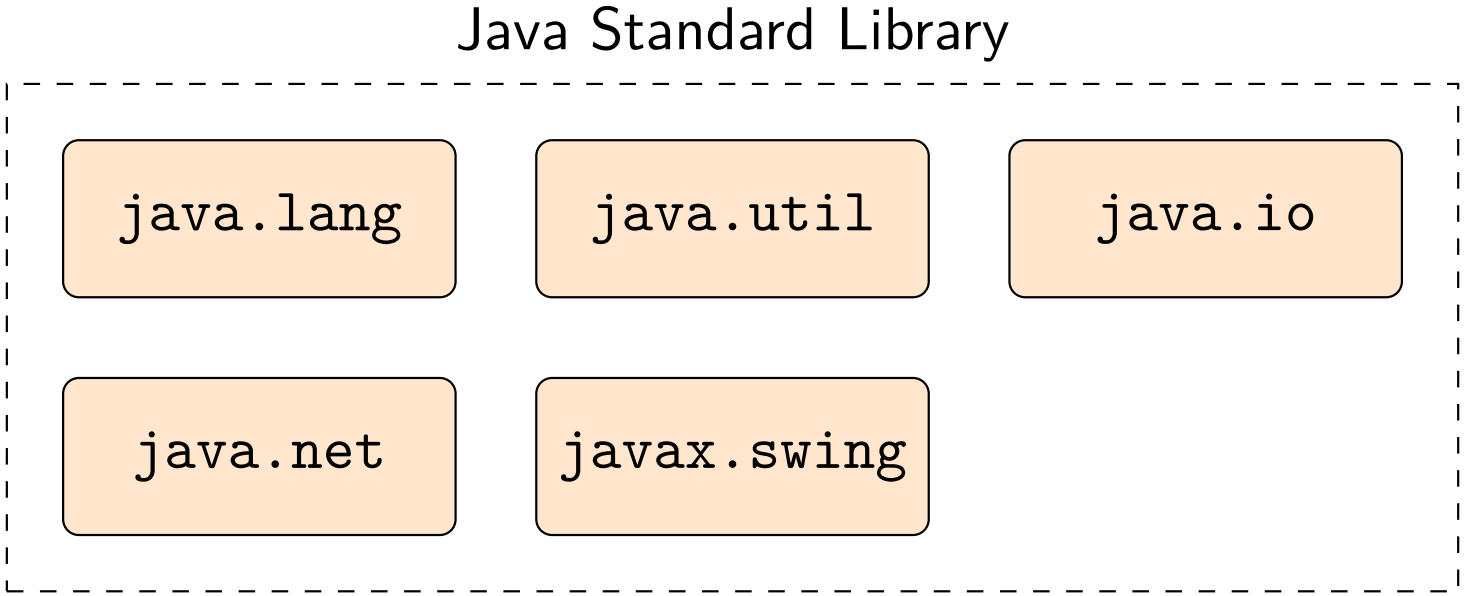
- The `package` statement declares a class's package and must be the file's first line.
- Folder structure on disk must exactly match the dotted package name.
- `import` lets code use short names for classes from other packages; default access is limited to the same package.

Course Overview & Today's Agenda

- Tour of Java's built-in packages: `java.lang`, `java.util`, ...
- Hands-on example using `ArrayList` and `Scanner`.
- Unit 3 wrap-up: inheritance, overriding, interfaces, abstract classes, and packages.
- Bridge to Unit 4: exception handling.

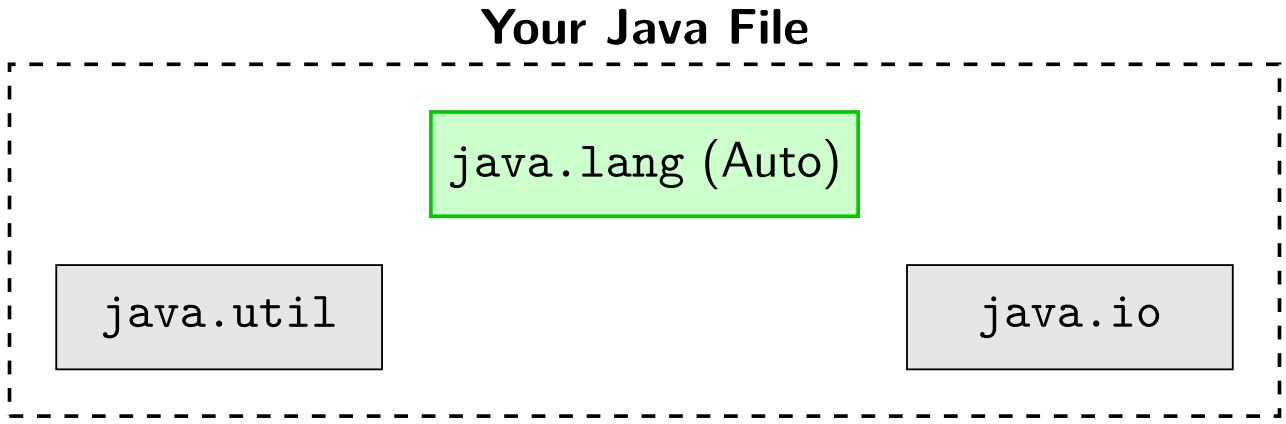
Built-in Packages: Why Java Ships With So Many

- Java comes with a huge standard library organised into built-in packages.
- Built-in packages follow the same `package` and `import` rules.
- Today's tour covers: `java.lang`, `java.util`, `java.io`, `java.net`, `java.awt`/`javax.swing`.



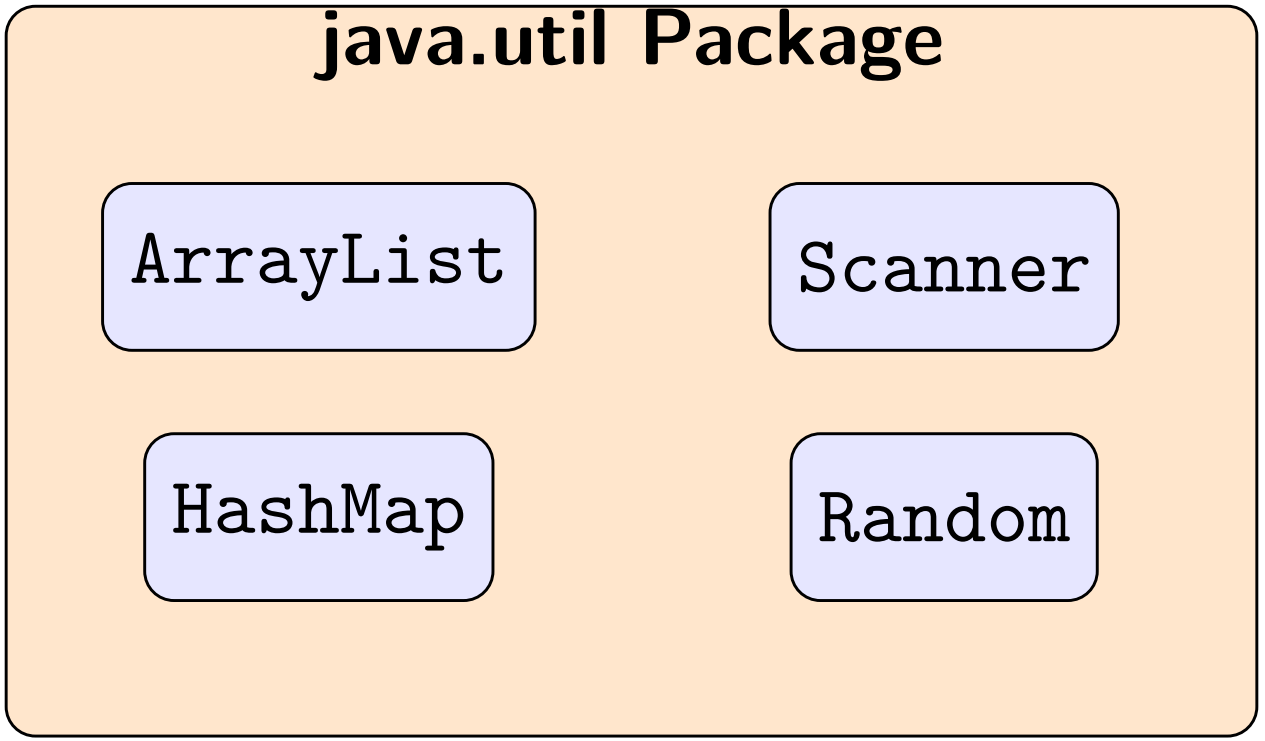
java.lang: Automatically Available

- java.lang is imported automatically into every Java file; no explicit import needed.
- Contains fundamental classes: String, Math, System, Object.
- This is why String and Math have been usable without import java.lang.String;.
- Every other package (java.util, java.io, etc.) DOES require an explicit import.



`java.util`: Utility and Collection Classes

- Holds general-purpose utility classes (e.g. Collections Framework).
- Contains `Scanner`, `Date/Calendar`, and `Random`.
- One of the most frequently imported packages in Java.

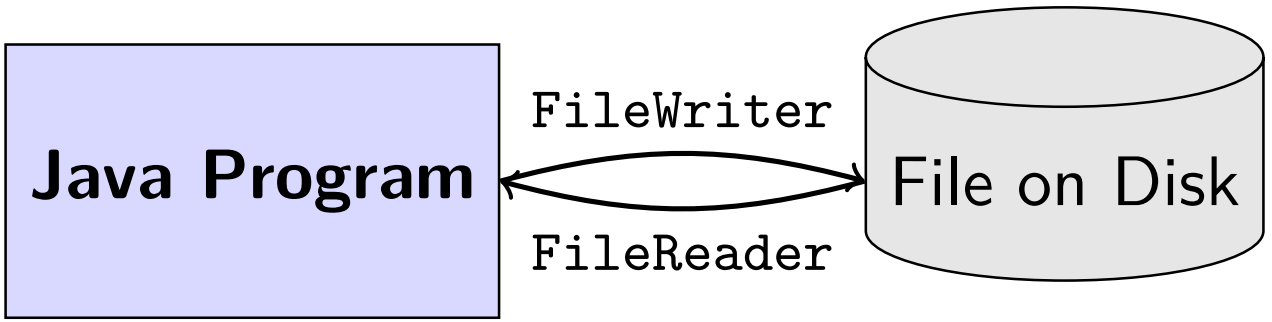


Hands-On: ArrayList and Scanner from java.util

```
import java.util.ArrayList;
import java.util.Scanner;

public class Demo {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Asha");
        names.add("Ravi");
        System.out.println(names);   // uses ArrayList's
            overridden toString()

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String input = sc.nextLine();
    }
}
```

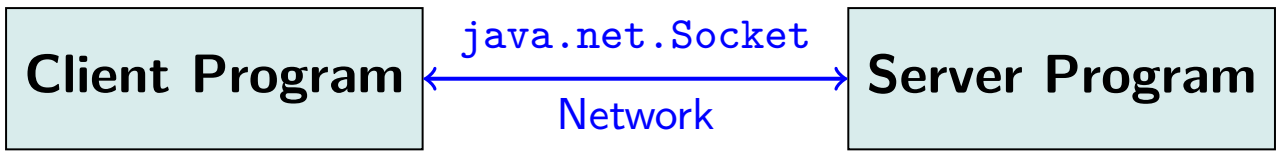
java.io: Input and Output

- Provides classes for reading/writing files and streams (`FileReader`, `FileWriter`).
- Used whenever a program needs to persist data to disk or read external files.
- Works alongside `Scanner`, which can also read from files.



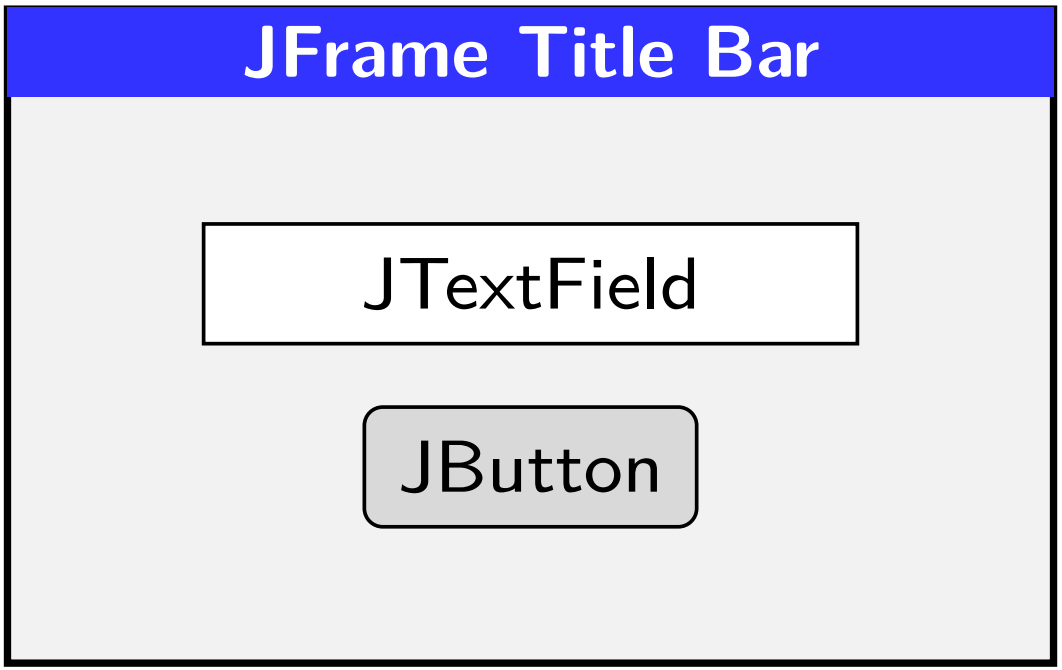
java.net: Networking

- Provides classes for network communication: `URL`, `Socket`, `ServerSocket`.
- Used to build programs that communicate over a network (client-server).
- Important to recognise by name for future networking-related coursework.



`java.awt` & `javax.swing`: GUI Programming

- Provide classes for building graphical user interfaces (windows, buttons).
- Swing components (`JFrame`, `JButton`) are preferred over older AWT components.
- GUI programming applies this unit's concepts: `JButton` inherits from a chain of superclasses.



Built-in Packages: Summary Table

Package	Purpose	Example Classes
`java.lang`	Core classes (auto-imported)	`String`, `Math`, `System`, `Object`
`java.util`	Utilities and collections	`ArrayList`, `Scanner`, `HashMap`
`java.io`	File and stream I/O	`FileReader`, `FileWriter`
`java.net`	Networking	`URL`, `Socket`, `ServerSocket`
`java.awt` / `javax.swing`	GUI components	`JFrame`, `JButton`, `JTextField`

Unit 3 Wrap-Up: The Big Picture

- **Inheritance**: `extends`, `super`, constructor chaining (Lectures 19-20).
- **Overriding**: redefining methods, dynamic dispatch, `Object` methods (Lectures 21-22).
- **Interfaces**: contracts via `implements`, safe multiple inheritance (Lectures 23-24).
- **Abstract Classes**: partial implementations combining approaches (Lecture 25).
- **Packages**: organising code with `package`/`import` (Lectures 26-27).

Common Mistakes and Exam Tips

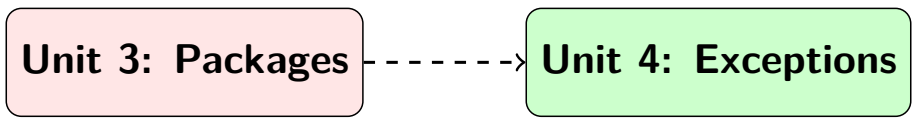
- **Mistake**: writing `import java.lang.String;` – unnecessary, since `java.lang` is automatic.
- **Mistake**: confusing `java.util`'s `Scanner` (input) with `java.io`'s classes (file I/O).
- **Exam Tip**: be ready to match each built-in package name to its purpose and example classes.
- **Exam Tip**: summarise Unit 3's five topics (3.1–3.5) in connected sentences, not just a list.

Key Takeaways

- `java.lang` is automatically imported and holds core classes like `String` and `Math`.
- `java.util` provides utilities (`ArrayList`, `Scanner`); `java.io` handles files; `javax.swing` handles GUIs.
- Unit 3 built a coherent progression: inheritance, overriding, interfaces, abstract classes, and packages.
- All built-in packages follow standard `package`/`import` mechanics.

Next Lecture Preview

- Lecture 28 opens Unit 4 with exception handling fundamentals.
- Introducing what exceptions are, the exception class hierarchy rooted in `Throwable`, and structured error handling.



Questions?

Recap: Unit 3 Wrap-Up (Packages)

- Lecture 27 covered built-in packages: `java.lang` (auto-imported), `java.util` (collections, `Scanner`), `java.io` (input/output).
- Unit 3 covered inheritance/`super`, overriding/`Object` class, interfaces, abstract classes, and packages.
- Today we shift from "organizing classes" (Unit 3) to "handling runtime failures" (Unit 4).

Course Overview & Today's Agenda

- Unit 4 roadmap: Exception handling (Lectures 28-31) $\rightarrow$ Multithreading (Lectures 32-36).
- What is an exception, and why does unhandled failure matter?
- The `Throwable` class hierarchy: `Error` vs `Exception`.
- Checked vs unchecked exceptions and common built-in types.

What Is an Exception?

Exception

An unwanted or unexpected event that disrupts the normal flow of a program's instructions during execution.

- Occurs at **runtime** (not compile time) – e.g., dividing by zero, accessing an invalid array index, or a missing file.
- Java represents every exception as an **object** – an instance of a class describing what went wrong.
- If not handled, an exception propagates up and causes abnormal program termination.

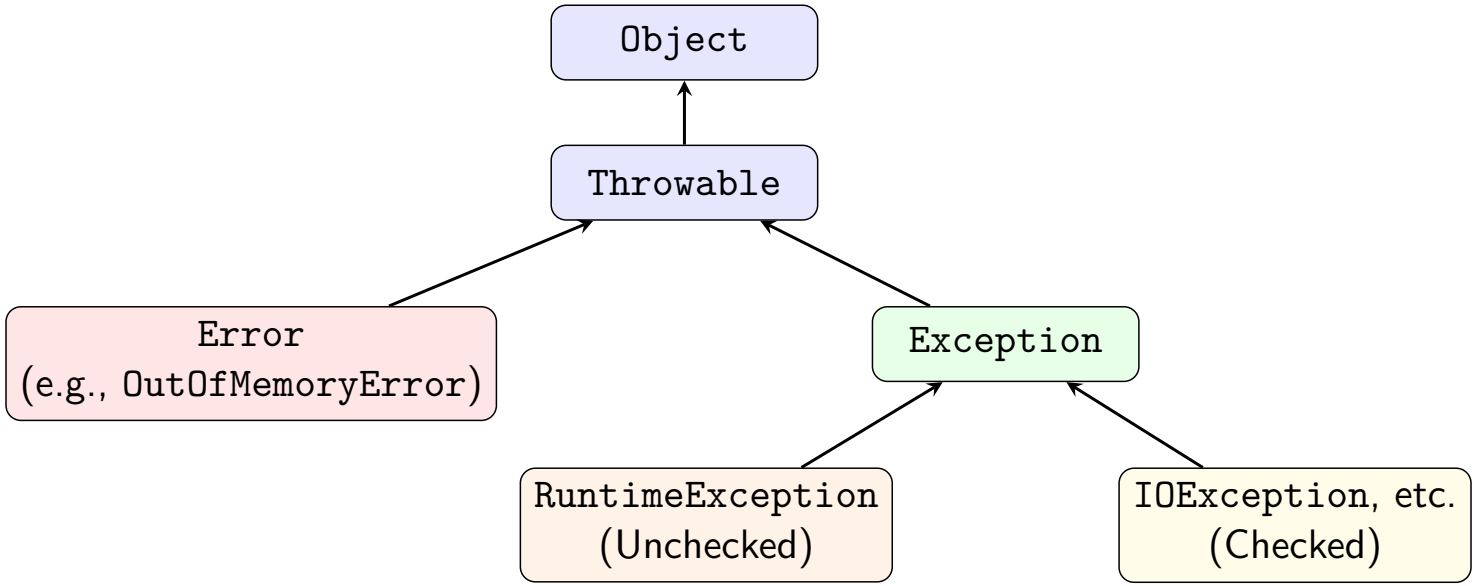
Example: A Program That Crashes

```java
public class DivideDemo {
    public static void main(String[] args) {
        int a = 10, b = 0;
        int result = a / b;    // ArithmeticException here
        System.out.println("Result = " + result);
    }
}
// Output: Exception in thread "main" java.lang.
    ArithmeticException: / by zero
// The println statement never runs - the program terminates
    abruptly
```

Why Exception Handling Matters

- Without handling, one bad input or edge case can crash an entire application.
- Handling exceptions lets the program respond gracefully – print a friendly message, log the problem, or recover and continue.
- **Critical for real-world software:** Banking systems, hospital software, or an autopilot cannot simply crash on bad input.
- Separates normal program logic from error-handling logic, keeping code readable.

The Throwable Class Hierarchy



- Every exception class in Java ultimately extends `Throwable`, which itself extends `Object`.

Error vs Exception

- Both extend `Throwable`, but have very different roles:

`Error`	`Exception`
Serious problems the application should **NOT** try to catch/recover from.	Conditions a well-written application **SHOULD** anticipate and handle.
Usually caused by environment/JVM running out of resources.	Caused by program logic, environment, or bad input.
Examples: `StackOverflowError`, `OutOfMemoryError`.	*Examples:* Invalid input, missing file, arithmetic problems.

Checked vs Unchecked Exceptions

Checked Exceptions	**Unchecked Exceptions**
Checked by the compiler at compile time.	**NOT** checked at compile time.
A method must either catch them or declare them with `throws`.	Compiler does not force handling.
'External, recoverable' conditions.	'Programming mistakes' surfacing at runtime.
Examples: `IOException`, `ClassNotFoundException`.	*Subclasses of `RuntimeException`:* `ArithmeticException`, `NullPointerException`.

Common Built-in Exception Classes

Exception Name	Category	Trigger Condition
`ArithmeticException`	Unchecked	Invalid arithmetic (e.g., division by zero)
`NullPointerException`	Unchecked	Method/field accessed on a null reference
`ArrayIndexOutOfBoundsException`	Unchecked	Array index is negative or $\geq$ array length
`NumberFormatException`	Unchecked	Invalid string-to-number conversion
`ClassNotFoundException`	Checked	Class not found on classpath at runtime
`IOException`	Checked	Input/output operation failed (e.g., file missing)

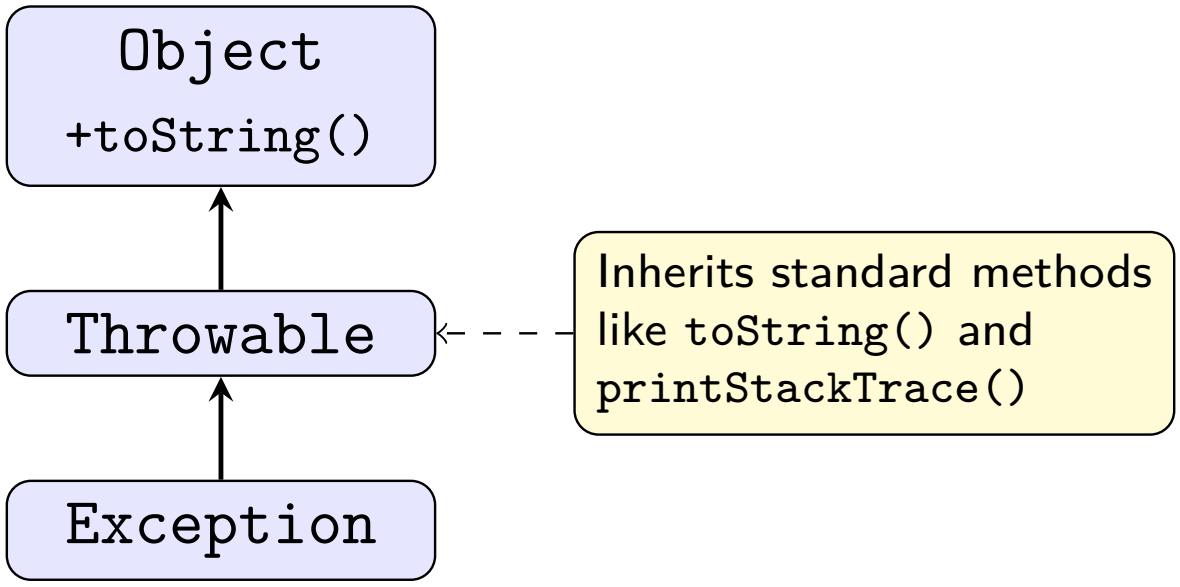
Example: NullPointer & ArrayIndexOutOfBounds

```
String name = null;
System.out.println(name.length());    // NullPointerException


int[] marks = {80, 90, 75};
System.out.println(marks[5]);         //
    ArrayIndexOutOfBoundsException
```

- `name.length()` fails because the object reference is null.
- `marks[5]` accesses an index beyond the array's valid range (0-2).

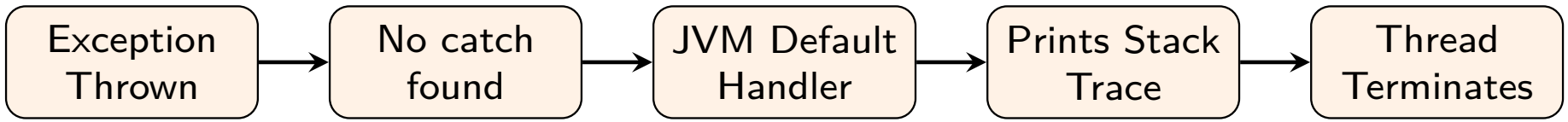
Exceptions Are Objects Too

- Every exception is an object (class extends `Throwable` → `Object`).
- Inherits `Object`'s methods: e.g., `toString()` is overridden to produce the `ClassName: message` text in stack traces.
- Can carry data via fields/constructors (foundation for custom exceptions).



How the JVM Handles an Unhandled Exception

- When an exception is thrown and no code catches it, the JVM's default handler takes over.



- The current thread terminates (stopping a single-threaded program).
- **Next:** Using `try/catch` to intercept this process instead.

Key Takeaways

- An exception is a runtime disruption to normal program flow, represented in Java as an object.
- The `Throwable` hierarchy splits into `Error` (unrecoverable JVM issues) and `Exception` (conditions to handle).
- Checked exceptions are verified by the compiler (must catch or declare); unchecked exceptions (`RuntimeException` subclasses) are not.
- Built-in exceptions (`ArithmeticException`, `NullPointerException`, etc.) each signal a specific failure condition.

Next Lecture Preview

Coming Up: Lecture 29

Lecture 29 puts this hierarchy to work by introducing the `try`, `catch`, and `finally` blocks, including how multiple catch blocks and exception propagation let a program actually recover from the failures introduced today.

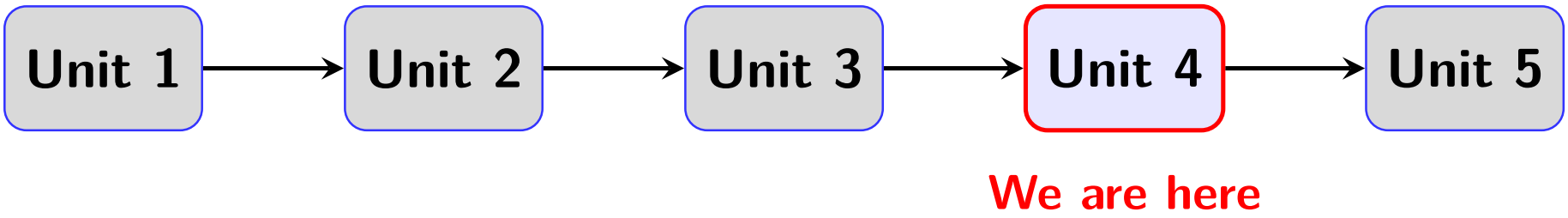
Questions?

Recap: Exception Hierarchy

- An exception is a runtime disruption, represented as an object descending from Throwable
- Error vs Exception; checked (compiler-enforced) vs unchecked (RuntimeException subclasses)
- Unhandled exceptions trigger the JVM's default handler, printing a stack trace and terminating the thread

Course Overview & Today's Agenda

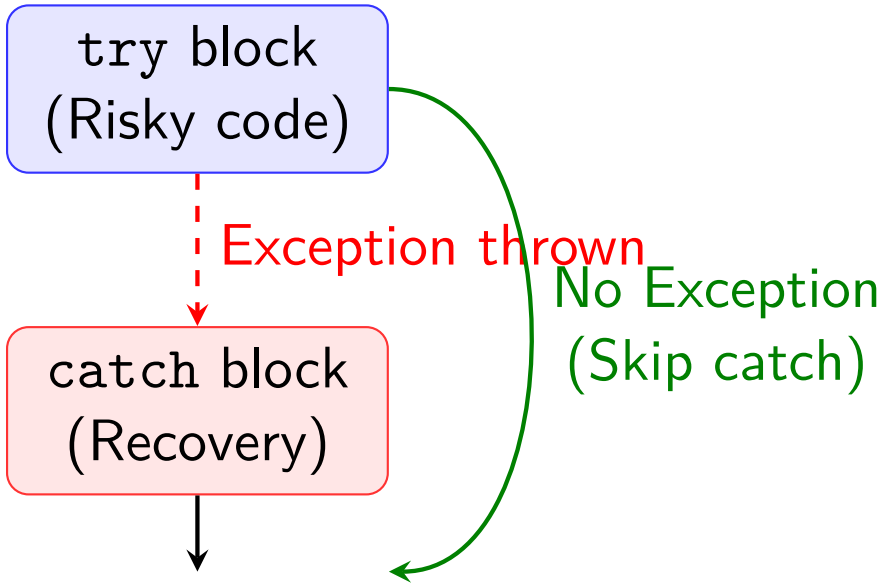
- The try-catch block: syntax and basic recovery
- The finally block: guaranteed cleanup code
- Multiple catch blocks and catch-order rules
- Exception propagation through the call stack



The try-catch Block: Syntax

- The JVM monitors the `try` block; jumps to `catch` on exception.
- If no exception, `catch` blocks are skipped entirely.

```
try {
    // risky code
    // exception occurs!
    // remaining skipped
} catch (ExceptionType e) {
    // handle it
}
```



Example: Fixing the Division-by-Zero Crash

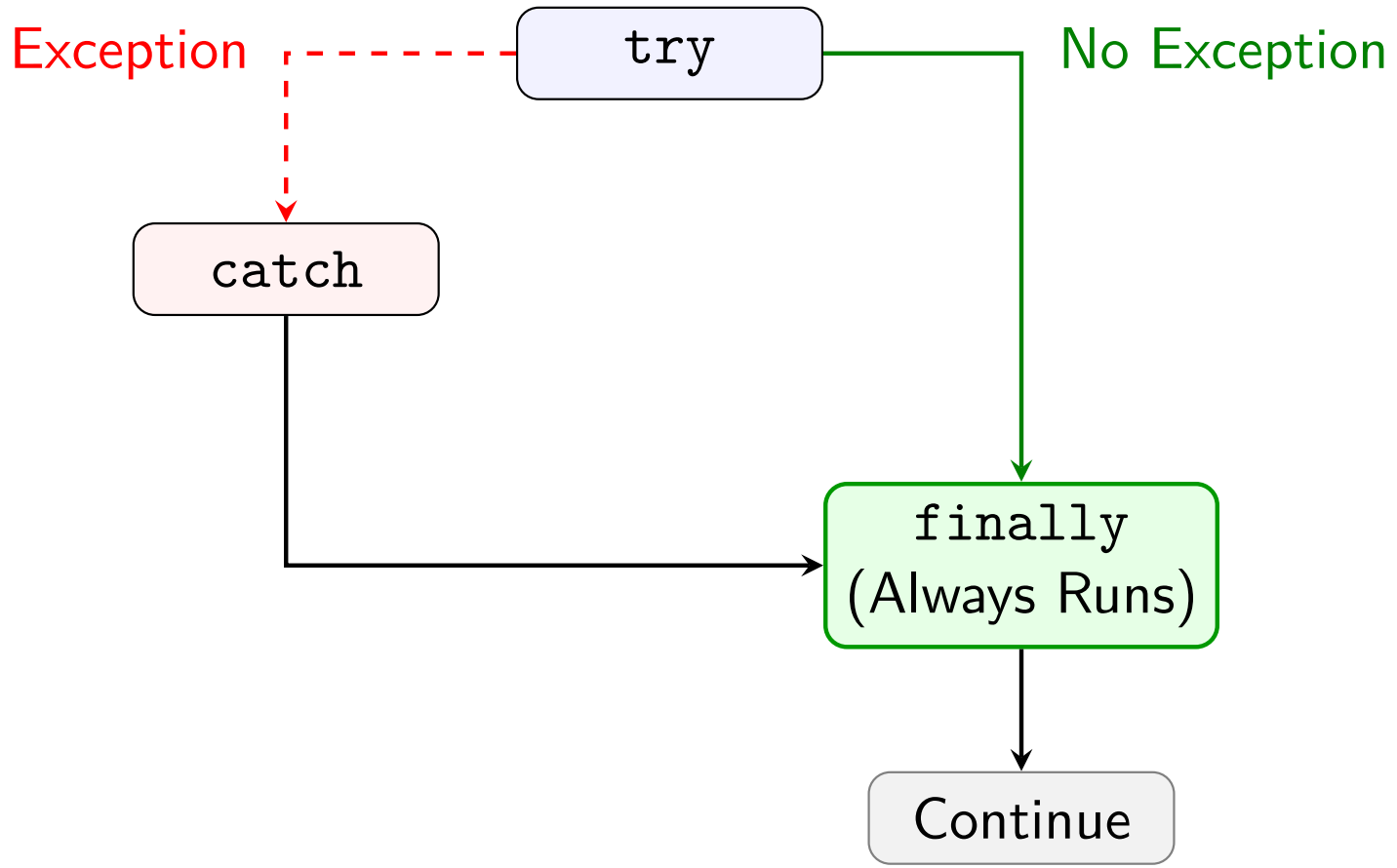
```
public class DivideDemo {
    public static void main(String[] args) {
        int a = 10, b = 0;
        try {
            int result = a / b;
            System.out.println("Result = " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e.
                getMessage());
        }
        System.out.println("Program continues normally");
    }
}
```

Output

Cannot divide by zero: / by zero
Program continues normally

The finally Block: Guaranteed Execution

- `finally` ALWAYS runs - regardless of exceptions.
- Runs even if `try` or `catch` contains a `return`.
- Typically used for cleanup (e.g. closing files, sockets).



Full Example: try-catch-finally

```
try {
    int[] marks = {80, 90, 75};
    System.out.println(marks[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Invalid index accessed");
} finally {
    System.out.println("Cleanup: closing resources");
}
```

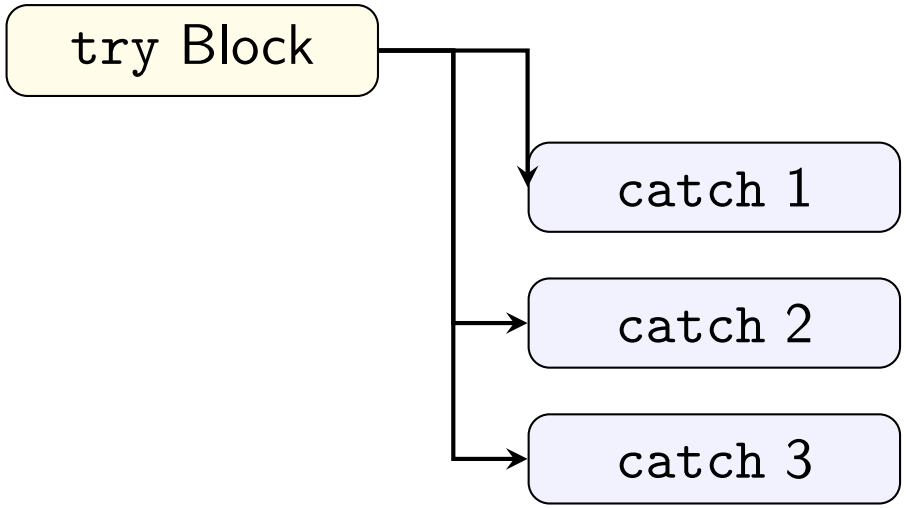
Output

Invalid index accessed
Cleanup: closing resources

Multiple Catch Blocks

- Only the FIRST matching `catch` block executes; the rest are skipped.

```
try {
    // ...
} catch (ArithmeticException e)
    {
    // handle math error
} catch (
    ArrayIndexOutOfBoundsException
     e) {
    // handle array error
} catch (Exception e) {
    // handle others
}
```



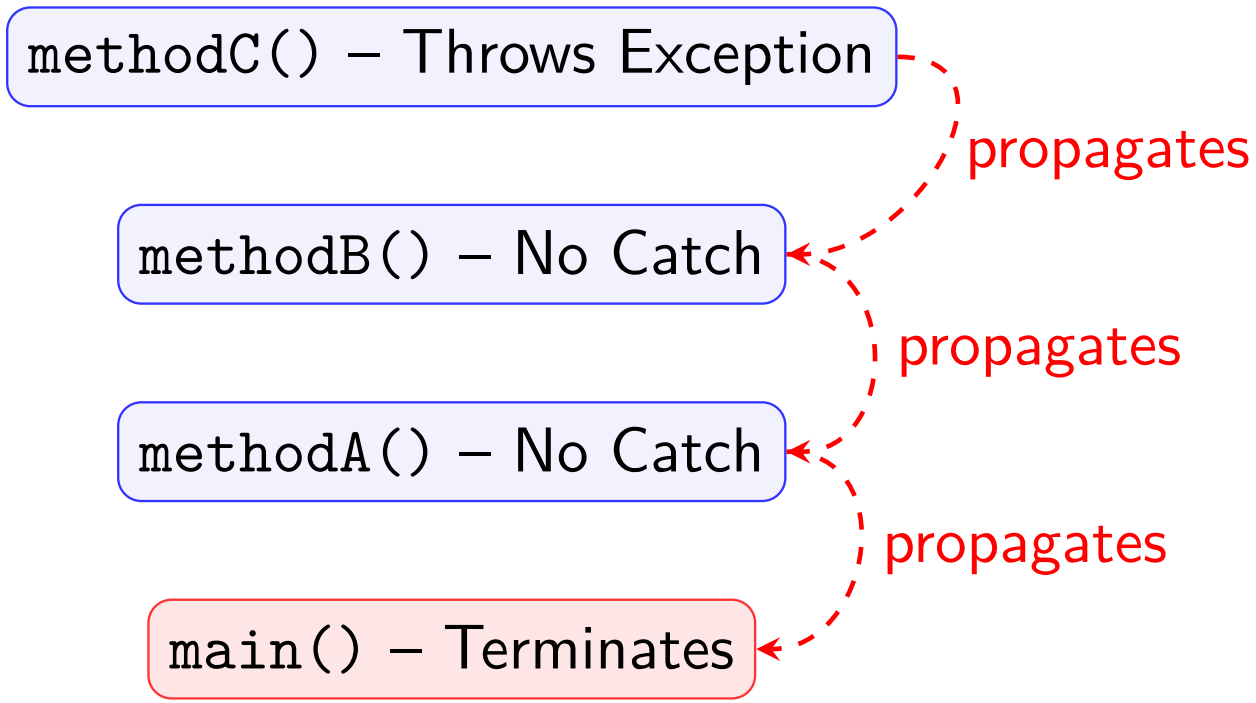
Catch Order Rule: Specific Before General

- **Rule:** Catch specific subclasses first, general superclasses (like `Exception`) last.
- **Reason:** Since `ArithmeticException` IS-A `Exception`, placing `Exception` first makes it always match, leaving specific blocks unreachable.

Valid Order	**Invalid Order (Compile Error)**
`catch (ArithmeticException e) { // ... }` `catch (Exception e) { // ... }`	`catch (Exception e) { // ... }` `catch (ArithmeticException e) { // ERROR: unreachable block }`

Exception Propagation: What Happens Without a Catch

- If uncaught, an exception **propagates** (passes up to the calling method).
- Bubbles up the call stack until a matching `catch` is found.
- If NO `catch` is found, it reaches JVM's default handler (program terminates).



Example: Exception Propagation Through Method Calls

```
public class PropagationDemo {
    static void methodC() { int x = 10 / 0; }          // throws
        here
    static void methodB() { methodC(); }                // does
        not catch
    static void methodA() {
        try {
            methodB();
        } catch (ArithmeticException e) {
            System.out.println("Caught in methodA: " + e.
                getMessage());
        }
    }
    public static void main(String[] args) { methodA(); }
}
```

methodC $\longrightarrow$ methodB $\longrightarrow$ methodA (catch!)

Common Mistakes with `finally`

★ Pitfalls to Avoid

- Putting a `return` statement inside `finally` silently overrides any return from `try`/`catch` - avoid this!
- Assuming `finally` will NOT run if there's a return in `try` - it **still runs** before the method returns.
- `System.exit()` inside `try`: `finally` is skipped entirely (a rare exception to the rule).
- Using `finally` for logic that changes program results rather than pure cleanup.

Key Takeaways

- `try` encloses risky code; `catch` handles a specific exception type; `finally` always executes for cleanup, exception or not.
- Multiple `catch` blocks respond differently per exception type, but must be ordered specific-to-general.
- Uncaught exceptions propagate up the call stack until a matching `catch` is found or the program terminates.
- The stack trace on an uncaught exception is literally a record of this propagation path.

Next Lecture Preview

- Lecture 30 turns to the `throw` and `throws` keywords:
  - How a programmer can deliberately raise an exception with `throw`.
  - How a method declares the checked exceptions it might pass upward using `throws`.

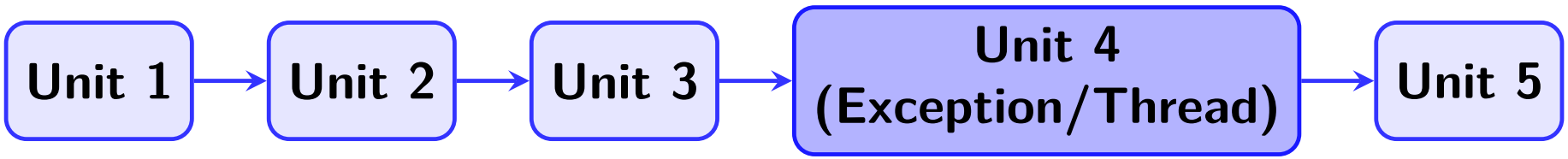
Questions?

Recap: try/catch/finally and Propagation

- **try/catch** handles exceptions; **finally** always executes for cleanup regardless of outcome.
- Multiple **catch** blocks must go from most specific to most general.
- Uncaught exceptions propagate up the call stack until caught.

Course Overview & Today's Agenda

- **throw**: deliberately raising an exception
- **throws**: declaring exceptions a method may pass on
- **throw vs throws**: critical distinction & compiler enforcement



The throw Keyword

- **throw** is used to explicitly raise an exception object from within code.
- Used when detecting invalid conditions that built-in JVM checks miss (e.g. business logic errors).
- Execution stops immediately; control transfers to nearest matching **catch** or propagates.

Syntax

```
throw new ExceptionClassName("message");
```

Example: Throwing an Exception for Invalid Input

```
public class AgeValidator {
    static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18
                or above");
        }
        System.out.println("Age accepted: " + age);
    }
    public static void main(String[] args) {
        checkAge(15);   // throws here
    }
}
// Output: Exception in thread "main" java.lang.
    IllegalArgumentException: Age must be 18 or above
```

Catching a Manually Thrown Exception

```
public static void main(String[] args) {
    try {
        checkAge(15);
    } catch (IllegalArgumentException e) {
        System.out.println("Validation failed: " + e.getMessage
            ());
    }
    System.out.println("Program continues");
}
```

The throws Keyword

- **throws** appears in a method signature declaring it might propagate checked exceptions.
- It does **NOT** handle the exception; it warns callers to catch or declare it.
- Enforced by compiler for **CHECKED** exceptions; unchecked may be declared optionally.

Syntax

```
returnType methodName(parameters) throws
ExceptionType1, ExceptionType2 { ... }
```

Example: Declaring and Handling a Checked Exception

```
import java.io.*;
public class FileDemo {
    static void readFile(String path) throws IOException {
        FileReader fr = new FileReader(path);   // may throw
            IOException
        // ... read logic ...
    }
    public static void main(String[] args) {
        try {
            readFile("data.txt");
        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        }
    }
}
```

Compile-Time Rule for Checked Exceptions

- If a method throws a checked exception, it **must** be caught with try/catch OR declared with throws.
- Otherwise, the code **fails to compile**.
- Unchecked exceptions (`RuntimeException`) are exempt from this rule.

```
Error: unreported exception java.io.IOException;
must be caught or declared to be thrown
```

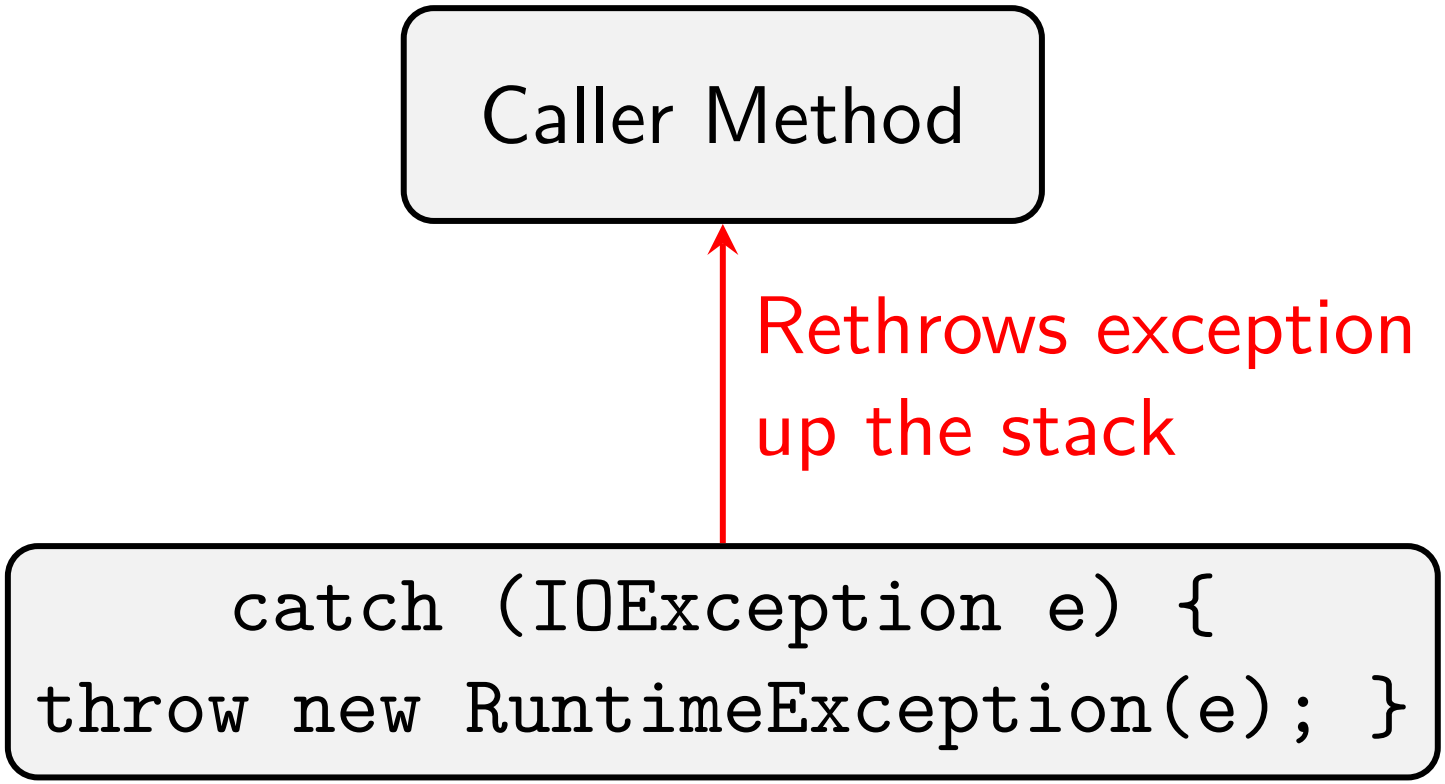
throw vs throws: Side-by-Side

throw	**throws**
A **statement** used **INSIDE** a method body.	A **declaration** used in the **METHOD SIGNATURE**.
Followed by a single exception **OBJECT** (e.g., `new Exception()`).	Followed by one or more exception **CLASS NAMES** (comma-separated).
Actually **raises/creates** the exception at that execution point.	Only **documents/declares** a possibility to compiler and caller.

- A method can have **throws** but never execute a **throw**, and vice versa for unchecked exceptions.

Rethrowing an Exception

- A **catch** block can use **throw** to re-raise an exception (often wrapped) to its caller.
- Useful for partial handling or logging before passing it up.



Example: Validating and Throwing on Bad Data

```
static double divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Divisor cannot be zero");
    }
    return (double) a / b;
}
```

Key Takeaways

- **throw**: A statement inside a method body that explicitly raises an exception object.
- **throws**: A signature declaration listing checked exceptions a method might pass to callers.
- Compiler enforces catch-or-declare **only** for checked exceptions.
- **throw** does the raising; **throws** documents the possibility. Conflating them is a common mistake!

Next Lecture Preview

- Lecture 31 builds on **throw** and **throws** to create custom exception classes tailored to specific application needs.
- We will also explore exception-handling best practices for writing robust Java code.

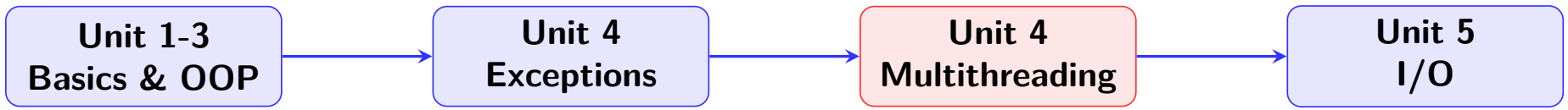
Questions?

Recap: throw and throws

- **throw** raises a specific exception object at a point in the code.
- **throws** is a method-signature declaration listing checked exceptions that might propagate to the caller.
- Checked exceptions must be caught or declared (compile-time rule); unchecked exceptions are exempt.

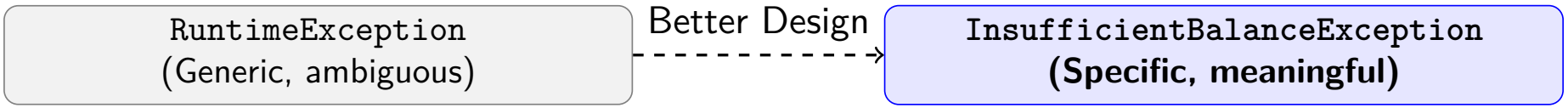
Course Overview & Today's Agenda

- Why create a custom exception class instead of reusing built-in ones.
- Steps to build a custom checked or unchecked exception.
- Worked examples: throwing and catching a custom exception.
- Exception handling best practices for real-world Java code.



Why Create a Custom Exception?

- Built-in exceptions (`ArithmeticException`, etc.) describe generic problems, not business rules.
- Gives a self-documenting name to a failure (e.g., `InsufficientBalanceException`).
- Carries extra domain data via custom fields and constructors.
- Callers can catch exact exceptions without hiding unrelated runtime errors.



Steps to Create a Custom Exception Class

1. Create a new class that extends `Exception` (checked) or `RuntimeException` (unchecked).
2. Provide at least one constructor that accepts a `String message` and passes it to the parent using `super(message)`.
3. Optionally add extra fields/constructors to carry additional domain-specific data.
4. Use `throw new YourException(...)` where invalid condition is detected.

```
class MyException extends Exception
+ MyException(String msg)
   super(msg);
```

Example: A Custom Checked Exception

```
public class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

public class BankAccount {
    double balance = 500.0;
    void withdraw(double amount) throws
        InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException("Insufficient
                funds for withdrawal of " + amount);
        }
        balance -= amount;
    }
}
```

Example: Catching the Custom Exception

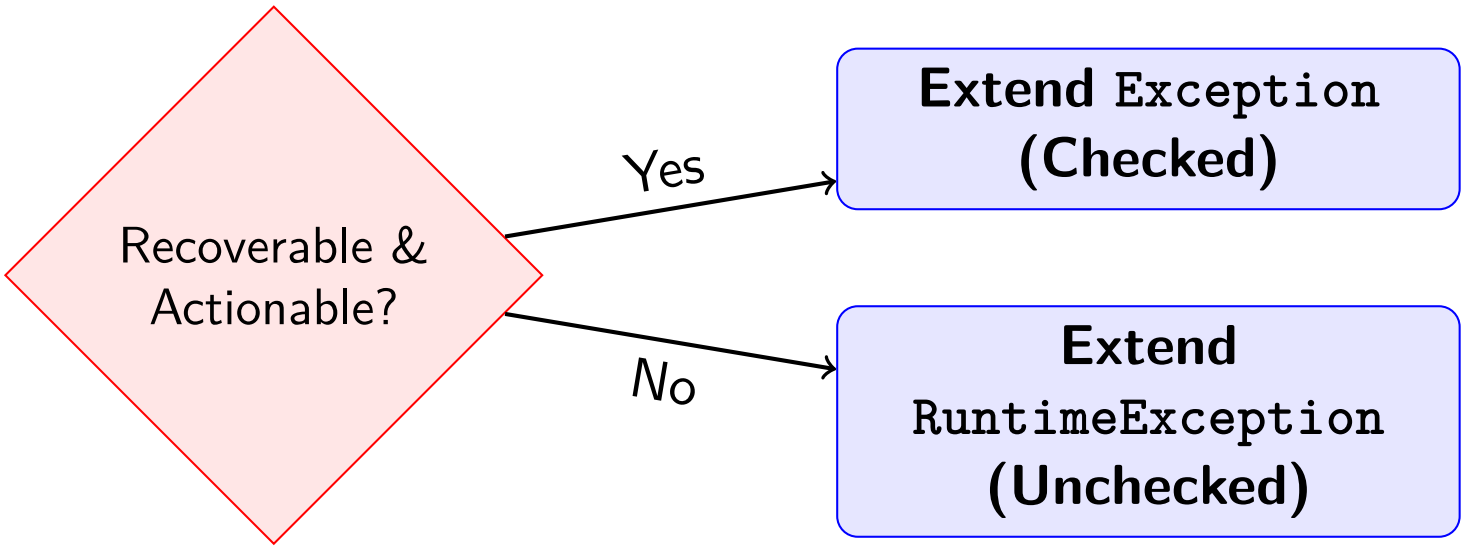
```
public class BankDemo {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        try {
            acc.withdraw(700.0);
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction failed: " + e.
                getMessage());
        }
    }
}
// Output: Transaction failed: Insufficient funds for withdrawal
      of 700.0
```

Example: A Custom Unchecked Exception

```
public class InvalidRollNumberException extends RuntimeException
      {
    public InvalidRollNumberException(String message) {
        super(message);
    }
}

static void validateRoll(int roll) {
    if (roll <= 0) {
        throw new InvalidRollNumberException("Roll number must
            be positive: " + roll);
    }
}
```

Checked vs Unchecked Custom Exception

Aspect	Extend `Exception` (Checked)	Extend `RuntimeException` (Unchecked)
Use Case	Expected, recoverable business conditions.	Programming mistakes or invalid usage.
Examples	Insufficient balance, invalid file format.	Invalid configuration value.
Caller Duty	Forced to catch or declare (`throws`).	Surfaces immediately, no clutter.
Caveat	Overusing clutters method signatures.	Can mask bugs if misused.



Best Practice: Catch Specific Exceptions

- Prefer catching the most specific exception type possible rather than a broad `catch (Exception e)`.
- Catching only what you expect avoids accidentally hiding unrelated bugs (e.g., `NullPointerException`).
- Use multiple specific catch blocks when different types need different logic.

```
// BAD: Hides all other errors
try {
  acc.withdraw(100);
} catch (Exception e) {
  // handles everything generically
}
```

```
// GOOD: Targeted handling
try {
  acc.withdraw(100);
} catch (InsufficientBalanceException e)
{
  // handles specific error
}
```

Best Practice: Never Swallow Exceptions Silently

- An empty catch block is a terrible anti-pattern: the error disappears with no trace.
- At minimum, log or print the exception (`e.getMessage()`, `e.printStackTrace()`).
- Silently swallowed exceptions cause bugs that are extremely difficult to trace.

```
// BAD: Empty catch block
try {
  doRiskyTask();
} catch (IOException e) {
  // ERROR SILENTLY IGNORED
}
```

```
// GOOD: Log or print
try {
  doRiskyTask();
} catch (IOException e) {
  e.printStackTrace();
  // OR log to a file
}
```

Best Practice: Cleanup with finally

- Always release resources (open files, DB connections) in a `finally` block so cleanup runs regardless of exceptions.
- Modern Java offers **try-with-resources** for `AutoCloseable` classes (automatic cleanup).
- Keep exception messages meaningful and specific (e.g., "Insufficient funds for withdrawal").

Exception Handling Checklist

✓ Meaningful & specific messages
✓ Catch specific exceptions
✓ Log or print (no swallowing)
✓ Cleanup resources in `finally`

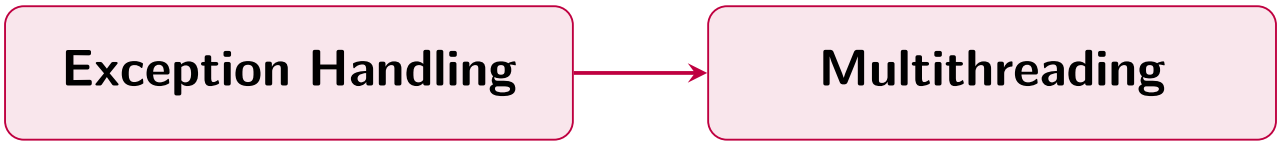
Key Takeaways

- **Custom exception:** Created by extending `Exception` (checked) or `RuntimeException` (unchecked), with a constructor calling `super(msg)`.
- **Checked vs Unchecked:** Checked for recoverable business conditions; unchecked for programming mistakes.
- **Best practices:** Catch specific types, never swallow silently, and always clean up resources.
- **Benefits:** Provide domain-specific, self-documenting names and carry extra data.

Next Lecture Preview

Coming Up: Multithreading

Lecture 32 opens the second half of Unit 4, moving from exception handling to multithreading - starting with what a thread is, the difference between a process and a thread, and why multithreading benefits real-world programs.



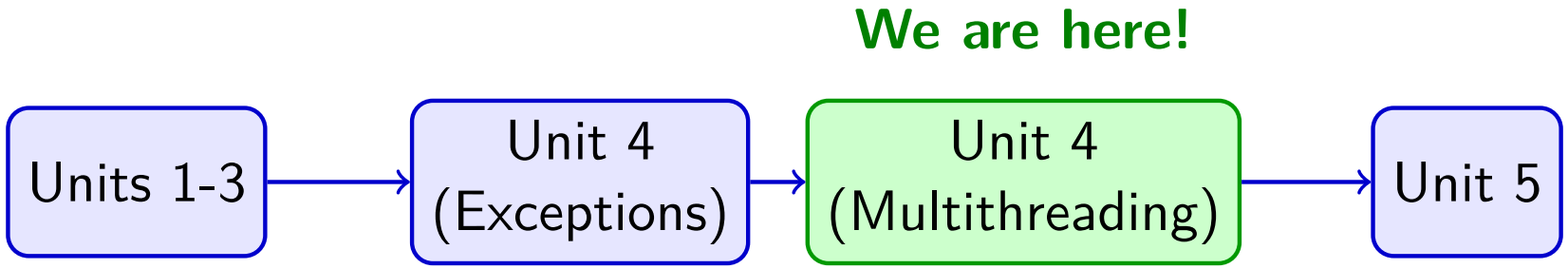
Questions?

Recap: Custom Exceptions & Best Practices

- Custom exceptions extend `Exception` (checked) or `RuntimeException` (unchecked), with `super(message)` in the constructor.
- Choose checked for recoverable business conditions, unchecked for programming-mistake-like failures.
- Best practices: catch specific types, never swallow exceptions silently, always clean up in `finally`.

Course Overview & Today's Agenda

- Unit 4 pivots today from exception handling to multithreading.
- What is a process? What is a thread?
- Process vs Thread: key differences.
- What is multithreading, and its core benefits.

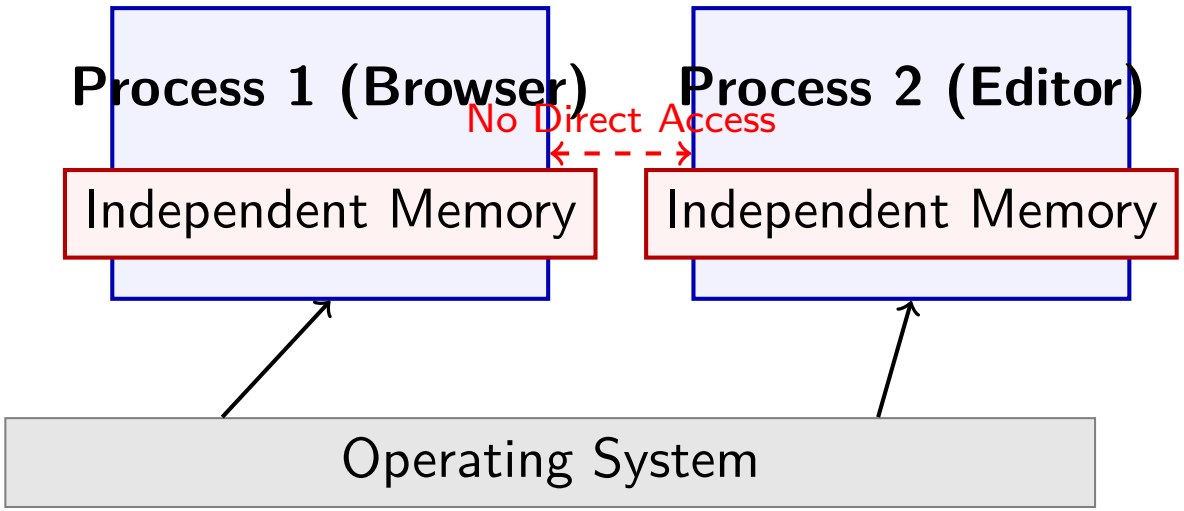


What Is a Process?

Process

An independent, running instance of a program, managed directly by the OS.

- Separate memory space; inaccessible to other processes.
- E.g. running browser and editor simultaneously.
- High creation cost due to independent OS memory.

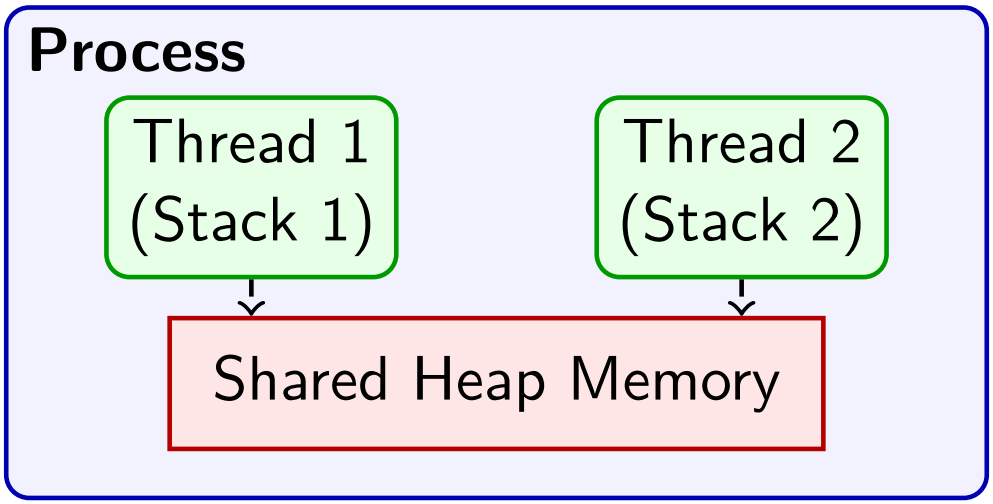


What Is a Thread?

Thread (Lightweight Process)

The smallest unit of execution within a process.

- A process contains multiple threads.
- Threads **SHARE** heap but have separate stacks.
- Cheaper/faster to create than a process.
- Java always runs at least the **main thread**.

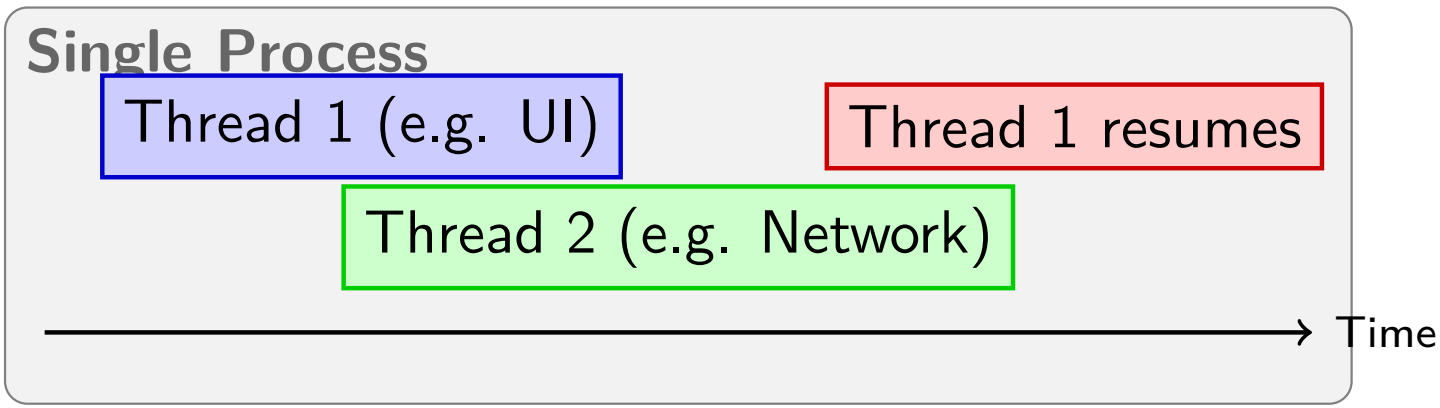


Process vs Thread

Feature	Process	Thread
Memory	Independent memory space	Shares memory (heap) with other threads
Creation Cost	Expensive (OS-level allocation)	Lightweight and fast
Communication	OS mechanisms (pipes, sockets)	Direct via shared memory/variables
Failure Isolation	Crashes don't affect other processes	Corrupting data can crash whole process

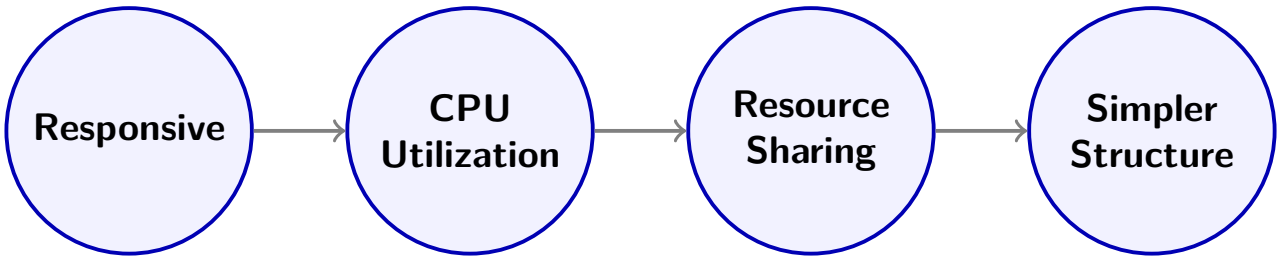
What Is Multithreading?

- Ability to run multiple threads concurrently within a single process.
- Threads execute independently but share process resources (memory, open files).
- CPU rapidly switches between threads (or runs in parallel on multi-core), creating simultaneous execution.
- Java has built-in support via `Thread` class and `Runnable` interface.



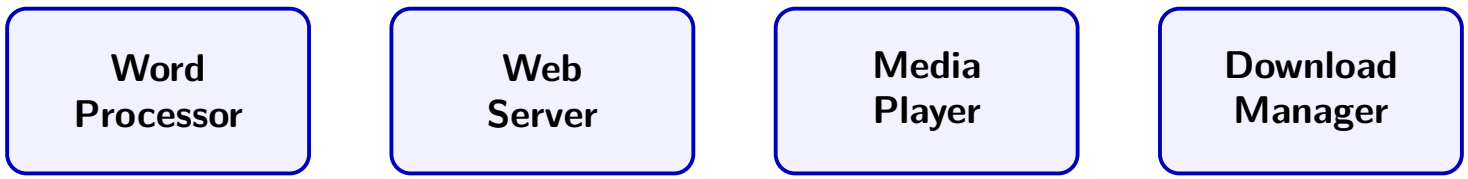
Benefits of Multithreading

- **Responsiveness:** A UI thread stays active while a background thread does heavy work (e.g., loading files).
- **Better CPU utilization:** Independent threads run in parallel on multi-core processors, using more available power.
- **Efficient resource sharing:** Threads share memory directly, avoiding inter-process communication overhead.
- **Simplified program structure:** Naturally concurrent tasks (e.g., handling client requests) map cleanly onto separate threads.



Real-World Examples

- **Word processor:** One thread handles typing/UI responsiveness; another auto-saves in the background.
- **Web server:** A separate thread handles each incoming client request (concurrent serving).
- **Media player:** One thread decodes/plays audio; another updates the progress bar and UI.
- **Download manager:** Multiple threads download different file parts simultaneously.



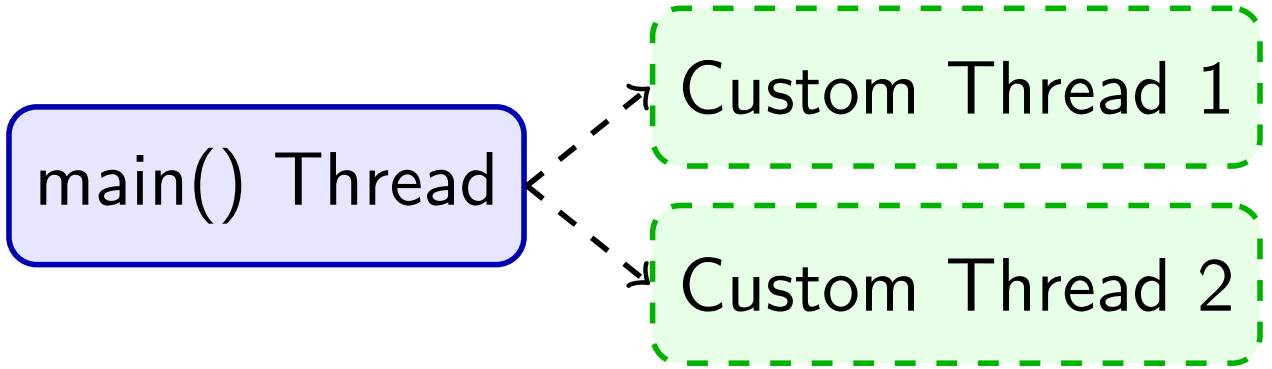
Multitasking vs Multithreading

Aspect	Multitasking	Multithreading
Basis	Process-based	Thread-based
Execution	OS runs separate processes concurrently	Single process runs multiple threads
Memory	Separate memory per process	Shared memory within process
Analogy	Browser & Music Player	Audio decoding & UI in Player

- Multithreading is a finer-grained form of multitasking (within one program vs across programs).

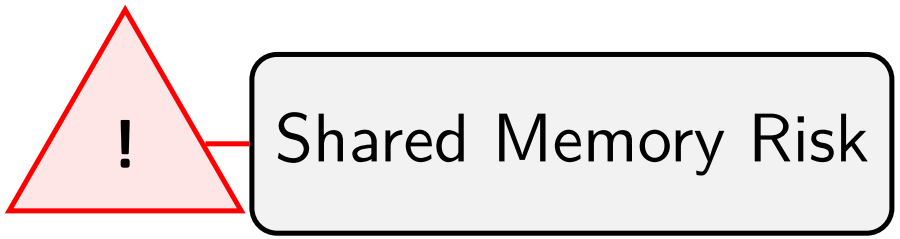
Java's Support for Multithreading

- Built-in support from the start in the `java.lang` package.
- **Two primary ways to create a thread:**
  1. Extending the `Thread` class.
  2. Implementing the `Runnable` interface.
- Every application automatically runs the **main thread** (executes `main()`).
- Crucial for Java's success in server-side/networked applications.



Multithreading Challenges

- Because threads share memory, simultaneous modification of data can cause inconsistent/corrupted results.
- **Synchronization** (coordinating access) is required but is beyond this unit's scope.
- Focus will remain on thread lifecycle and creation, not synchronization mechanics.



Key Takeaways

- **Process vs Thread:** Process is independent with own memory; thread is lightweight, sharing process memory.
- **Multithreading:** Running multiple threads concurrently within a single process.
- **Benefits:** Responsiveness, CPU utilization, efficient resource sharing, natural fit for concurrent tasks.
- **Java:** Built-in support via `Thread` and `Runnable`; `main` thread exists by default.

Next Lecture Preview

Lecture 33: Thread Lifecycle

Examines the thread lifecycle in detail - the New, Runnable, Running, Blocked/Waiting, and Terminated states a thread passes through, and the methods that trigger each transition.

Questions?

Recap: Process vs Thread and Multithreading Benefits

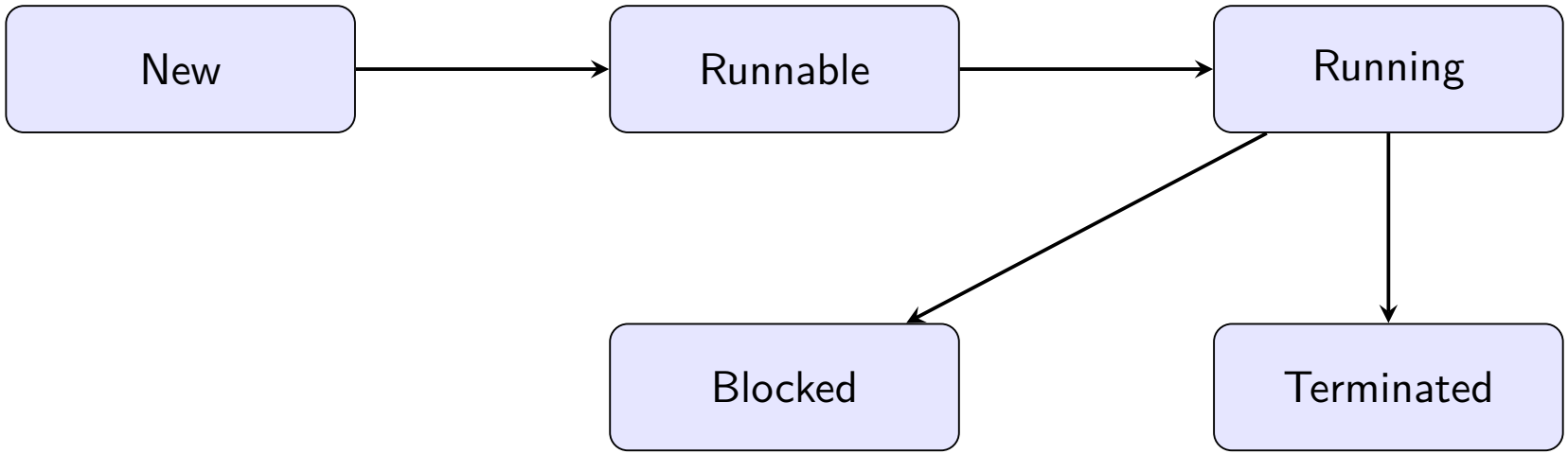
- Process has independent memory; thread is a lightweight unit of execution sharing its process's memory
- Multithreading runs multiple threads concurrently within one process for better responsiveness and CPU utilization
- Java has built-in multithreading support via `Thread` and `Runnable`, with every program already running a main thread

Course Overview & Today's Agenda

- The five states of the Java thread lifecycle
- What causes a thread to move from one state to the next
- Key methods: `start()`, `sleep()`, `wait()`, `notify()`, `join()`
- Observing thread state in code with `getState()`

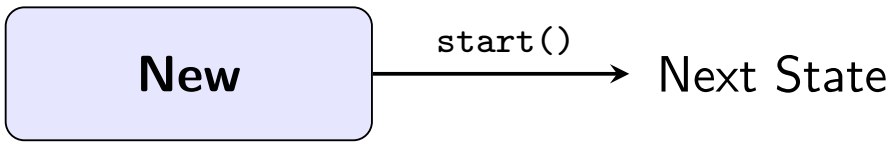
Thread Lifecycle: Overview of Five States

- Every Java thread moves through five states: New, Runnable, Running, Blocked/Waiting, and Terminated
- A thread object exists in exactly one of these states at any given moment
- `Thread.State` enum represents these, retrievable via `getState()`



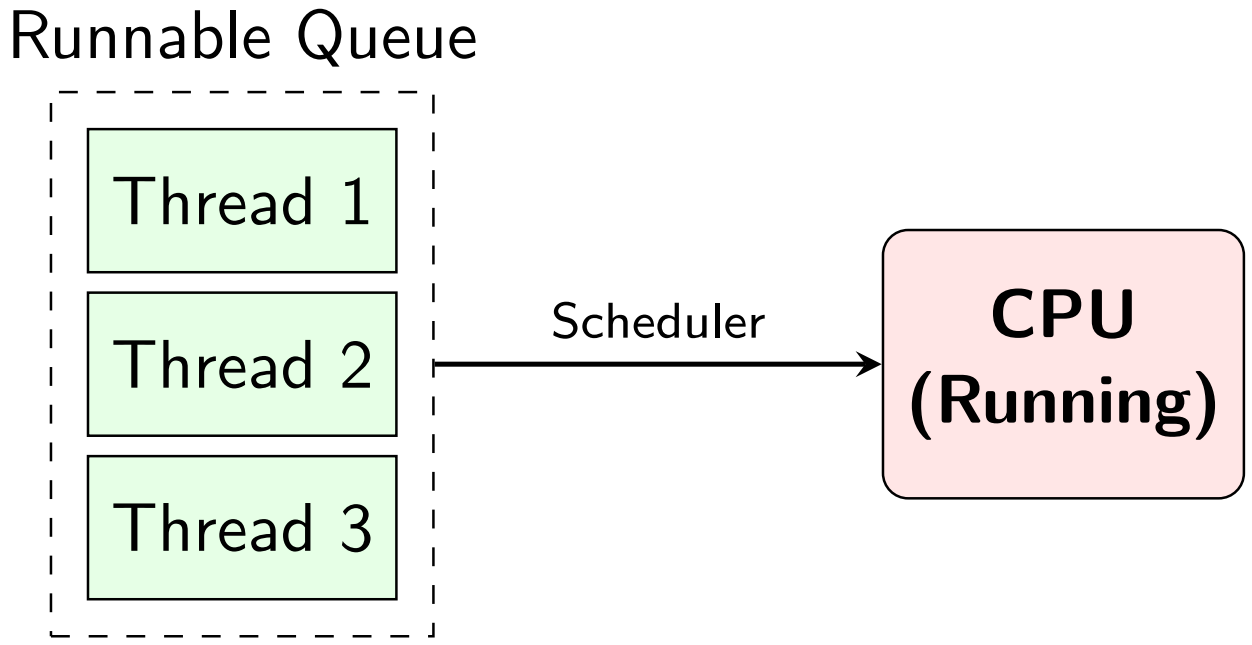
State 1: New

- Enters New state when created: `Thread t = new Thread();`
- Object exists in memory; underlying OS thread NOT yet started
- No code inside `run()` has executed
- Moves out of New only when `start()` is called



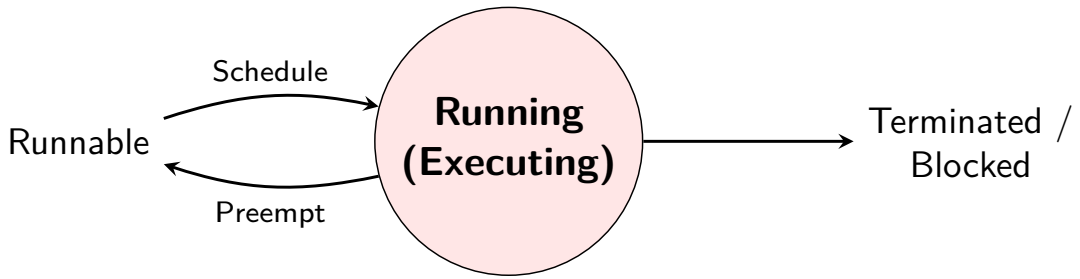
State 2: Runnable

- After `start()`, enters Runnable; eligible to run, waiting for CPU
- May not be executing instantly; just ready in queue
- Scheduler decides who gets CPU next
- Multiple threads can be Runnable simultaneously



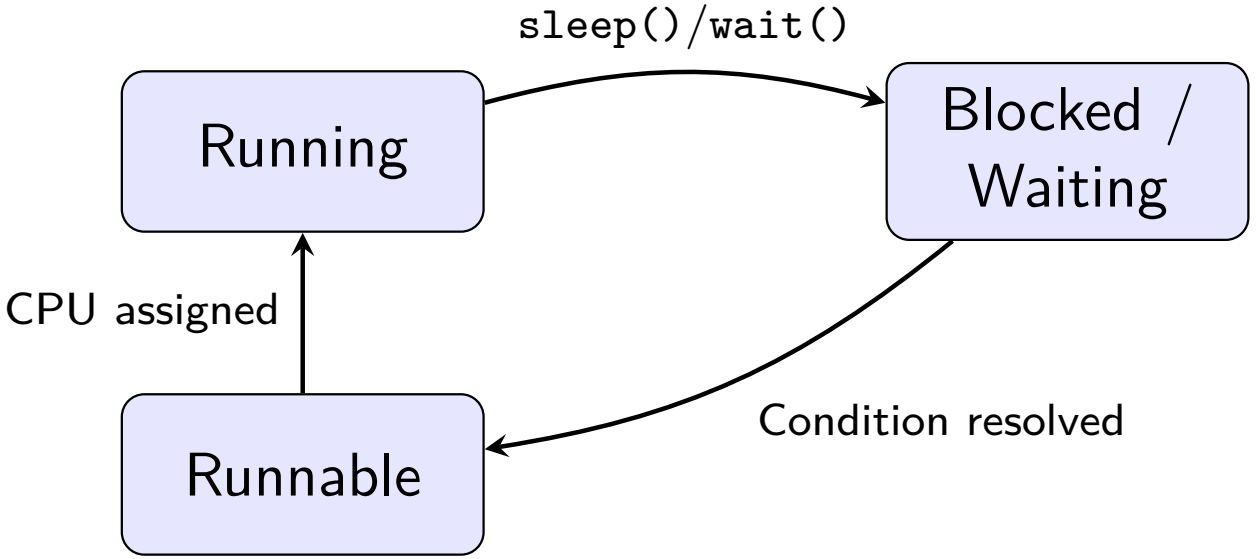
State 3: Running

- Entered when scheduler allocates CPU; `run()` code executes
- Only one thread per core can be Running at a time
- Preempted threads move back to Runnable
- Can also move to Blocked/Waiting or Terminated



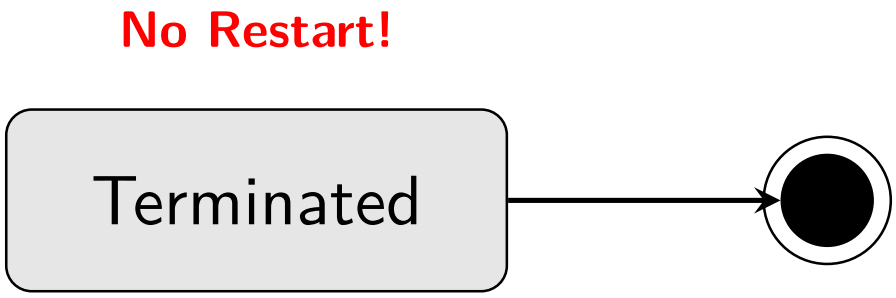
State 4: Blocked / Waiting

- Entered when unable to continue (`sleep()`, `wait()`, locks)
- Alive but not eligible to run until condition is resolved
- ALWAYS returns to Runnable, never directly to Running
- Useful to deliberately pause progress without terminating

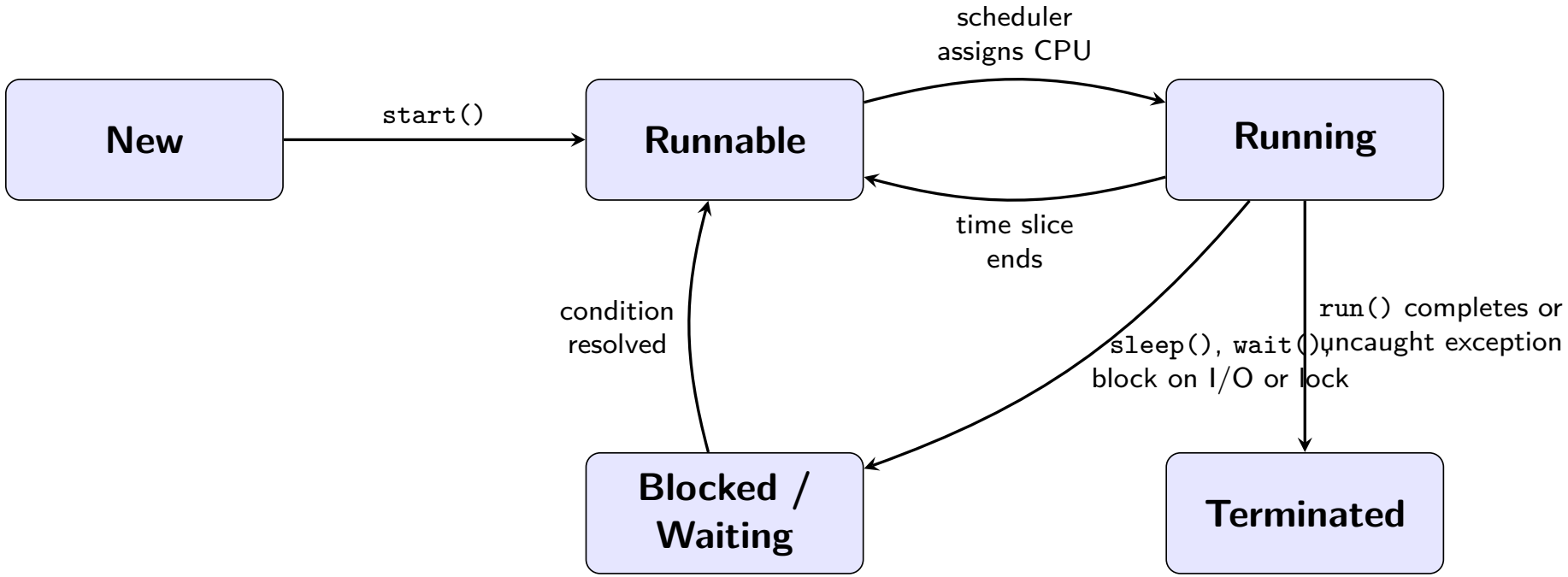


State 5: Terminated (Dead)

- Entered when `run()` completes normally or via exception
- Cannot be restarted; `start()` throws `IllegalThreadStateException`
- Final state; object exists but is no longer an active thread



Full Thread Lifecycle Diagram



Methods That Trigger State Transitions

Method / Event	**Resulting Transition**
`start()`	New → Runnable
`Thread.sleep(ms)`	Running → Blocked/Waiting (for duration)
`wait()`	Running → Waiting (until `notify()`)
`join()`	Calling thread waits until target terminates
`run()` completes	Running → Terminated

Example: Observing Thread State with getState()

```
Thread t = new Thread(() -> {
    try { Thread.sleep(1000); } catch (InterruptedException e) {
          }
});
System.out.println(t.getState());    // NEW
t.start();
System.out.println(t.getState());    // RUNNABLE (or
    TIMED_WAITING if sleeping already)
t.join();
System.out.println(t.getState());    // TERMINATED
```

Key Takeaways

- Every thread passes through 5 states: New, Runnable, Running, Blocked/Waiting, Terminated
- `start()` moves thread to Runnable; scheduler promotes to Running
- `sleep()`, `wait()`, and locking move thread to Blocked/Waiting (always returns to Runnable)
- Terminated threads cannot be restarted

Next Lecture Preview

Lecture 34: Thread Creation

We move from thread states to thread creation, comparing the two ways Java lets a programmer build a thread:

- Extending the `Thread` class
- Implementing the `Runnable` interface

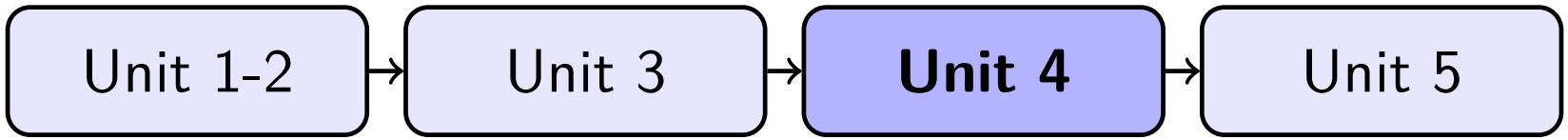
Questions?

Recap: Thread Lifecycle

- Every thread passes through New, Runnable, Running, Blocked/Waiting, and Terminated
- `start()` moves thread from New to Runnable; scheduler promotes to Running
- `sleep()`/`wait()` move Running to Blocked/Waiting; returns to Runnable

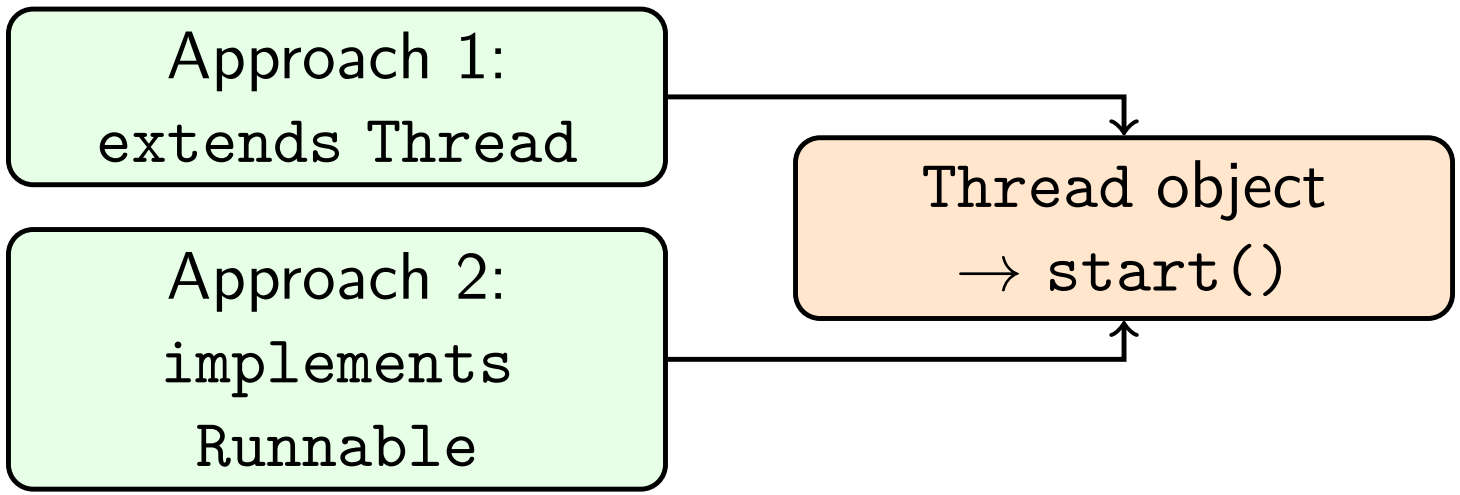
Course Overview & Today's Agenda

- Two ways to create a thread: an overview
- Extending `Thread` vs implementing `Runnable`
- The critical difference between `start()` and `run()`



Two Ways to Create a Thread in Java

- **Approach 1:** Create a class that `extends` `java.lang.Thread` and override `run()`
- **Approach 2:** Create a class that `implements` `java.lang.Runnable`, define `run()`, pass to a `Thread` object



Approach 1 Preview: Extending Thread

Preview - full detail in Lecture 35

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running via extends
            Thread");
    }
}
// Usage:
MyThread t = new MyThread();
t.start();
```

Approach 2 Preview: Implementing Runnable

Preview - full detail in Lecture 36

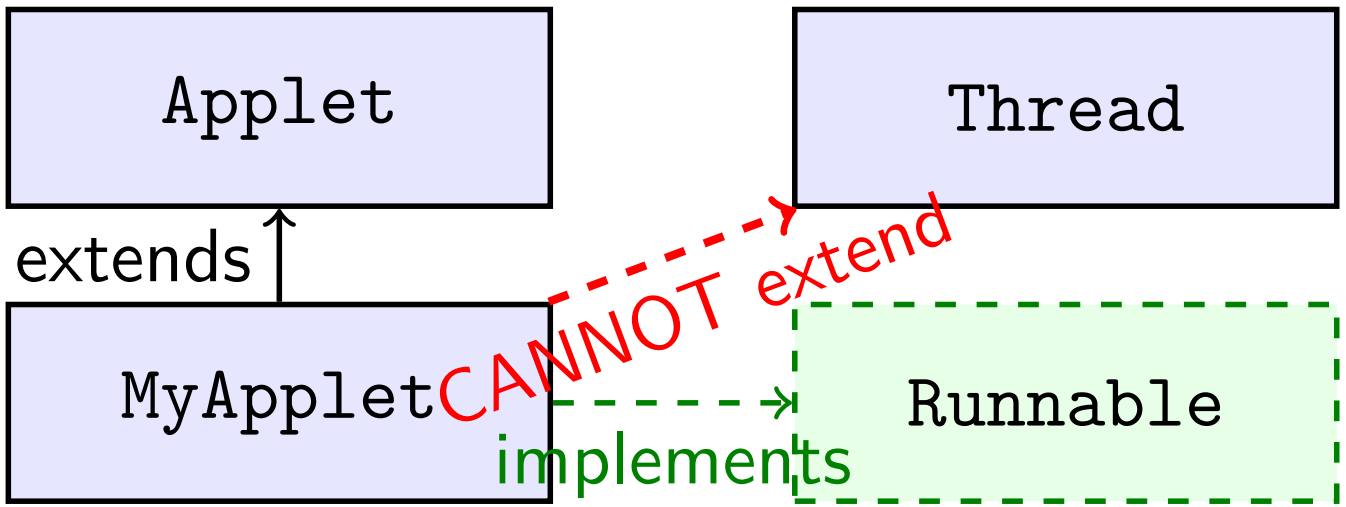
```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task running via implements
            Runnable");
    }
}
// Usage:
MyTask task = new MyTask();
Thread t = new Thread(task);
t.start();
```

Extending Thread vs Implementing Runnable

Extends `Thread`	**Implements `Runnable`**
Class *becomes* a Thread (IS-A)	Class provides a task (HAS-A)
Cannot extend another class (single inheritance constraint)	Free to extend another class
Each thread needs a new object	Same `Runnable` object can be shared
Both use `run()` to define task code	

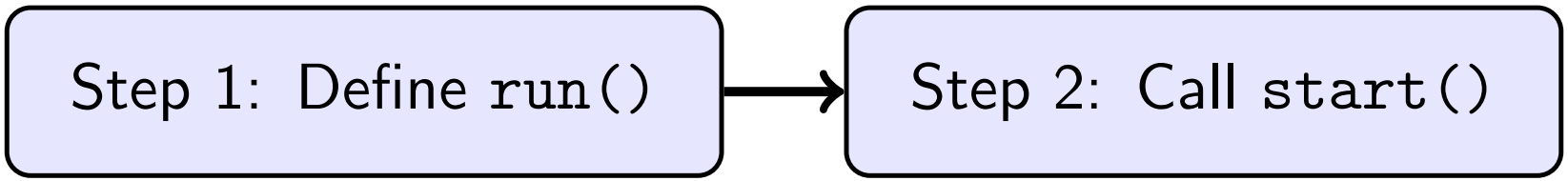
Why Runnable Is Generally Preferred

- Single inheritance: a class extending another cannot also extend `Thread`
- `Runnable` interface keeps class free to extend something else
- Separates 'the task' from 'the mechanism that runs it'



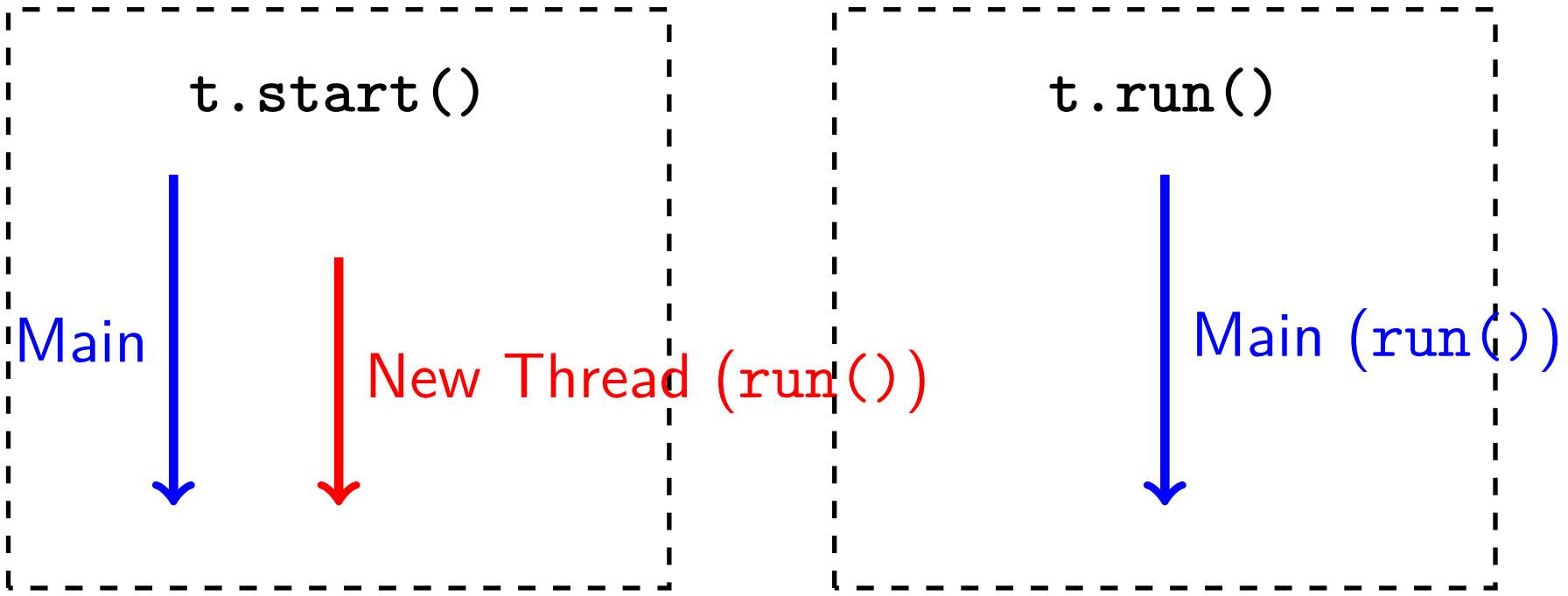
The Common Two Steps in Both Approaches

- **Step 1:** Define code thread should run by overriding/implementing `run()`
- **Step 2:** Obtain a `Thread` object and call `start()`



Critical Distinction: start() vs run()

- `t.start()`: creates genuinely new thread, which then calls `run()`
- `t.run()`: does NOT create new thread; executes sequentially on same thread



Example: start() vs run() in Practice

```
MyThread t1 = new MyThread();
// WRONG for concurrency: runs on main thread, sequentially
t1.run();

MyThread t2 = new MyThread();
// CORRECT: JVM creates a new thread, which executes run()
t2.start();
```

Output prediction

`t1.run()` runs in "main" thread.

`t2.start()` runs in a newly spawned thread (e.g., "Thread-0").

Choosing a Thread Creation Approach

Requirement / Scenario	**Approach to Use**
Class needs to extend another class	`implements Runnable`
Share one task object across threads	`implements Runnable`
Simplest setup, no inheritance needs	Either (`extends` is simpler)
Actually launch a concurrent thread	Always call `start()`

Key Takeaways

- Two approaches: extend `Thread` or implement `Runnable`
- `Runnable` preferred for flexibility (keeps inheritance slot free)
- Both approaches define task via `run()` and launch via `start()`
- `run()` alone doesn't create a thread; `start()` achieves concurrency

Next Lecture Preview

- Lecture 35 goes deep on the first approach - extending the `Thread` class
- Full worked examples of overriding `run()` and starting multiple thread objects
- Useful `Thread` methods like `getName()` and `sleep()`

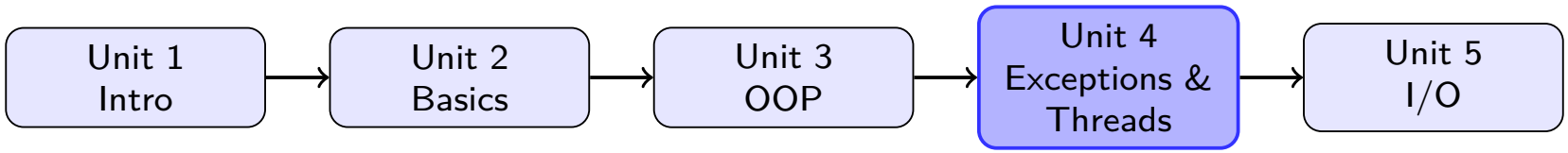
Questions?

Recap: Thread Creation Approaches Overview

- Java offers two thread creation approaches: extending `Thread`, or implementing `Runnable`
- Extending `Thread` uses up the single-inheritance slot; implementing `Runnable` keeps the class free to extend something else
- Calling `start()` launches a genuine new thread; calling `run()` directly is just a normal method call on the current thread

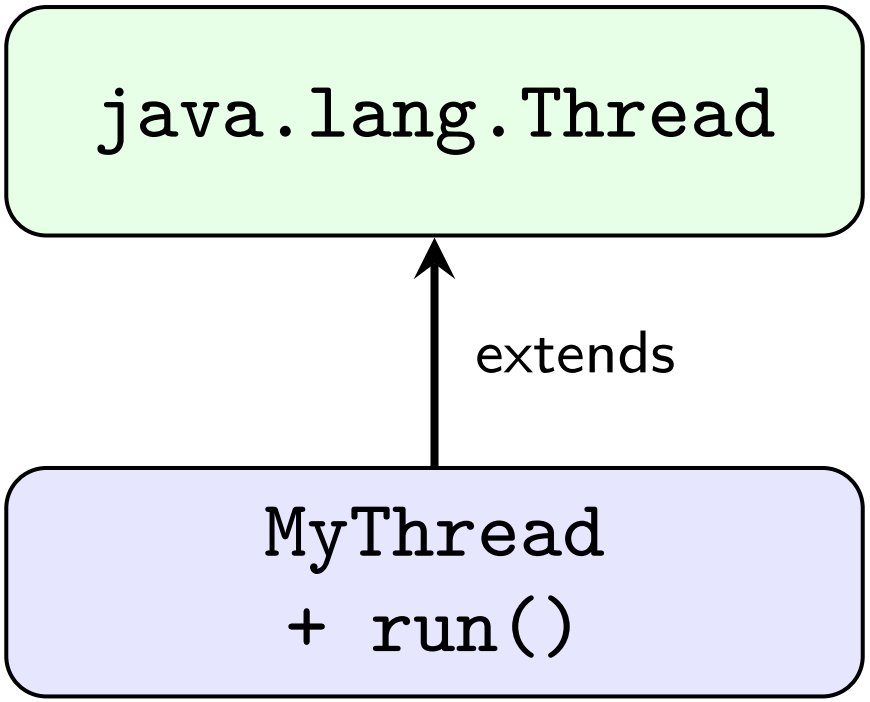
Course Overview & Today's Agenda

- Steps to create a thread by extending `Thread`, with full worked example
- Running multiple thread objects concurrently
- Useful `Thread` methods: `getName()`, `setName()`, `getId()`, `currentThread()`
- Using `sleep()` inside `run()` and handling `InterruptedException`



Steps to Extend the Thread Class

- 1. Define a class that extends `java.lang.Thread`
- 2. Override `run()` method and place code to execute inside it
- 3. Create an object of this subclass
- 4. Call `start()` on that object (never `run()` directly)

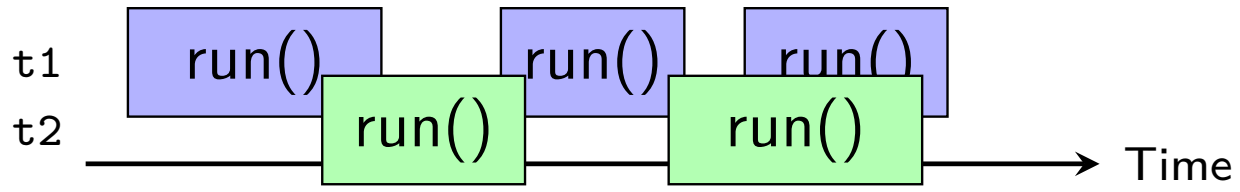


Example: A Basic Thread Subclass

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
        System.out.println("Main thread continues...");
    }
}
```

Running Multiple Thread Objects Concurrently

```
public class MultiThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
// Two independent threads now execute the same run() logic
    concurrently
// Their Count: output lines may interleave unpredictably
```



Useful Thread Methods: Naming and Identity

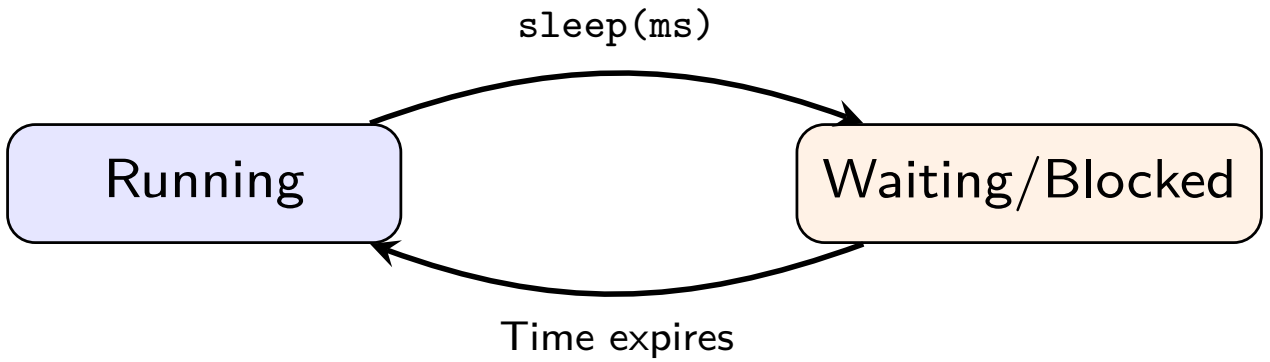
- `getName()` - returns thread name (default is 'Thread-0', etc.)
- `setName(String)` - assigns custom, meaningful name
- `getId()` - returns unique long identifier assigned by JVM
- `Thread.currentThread()` - static method returning reference to currently executing thread

Example: getName() and currentThread()

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running in: " + Thread.currentThread
            ().getName());
    }
}
public class NameDemo {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.setName("WorkerThread");
        t.start();
        System.out.println("Running in: " + Thread.currentThread
            ().getName());
    }
}
// Output (order may vary):
// Running in: WorkerThread
// Running in: main
```

The sleep() Method

- `Thread.sleep(ms)` pauses CURRENTLY EXECUTING thread
- Static method, e.g. `Thread.sleep(1000)` for a 1-second pause
- Moves thread to Blocked/Waiting state, returns to Runnable when time elapses
- Declares checked exception `InterruptedException` which MUST be caught

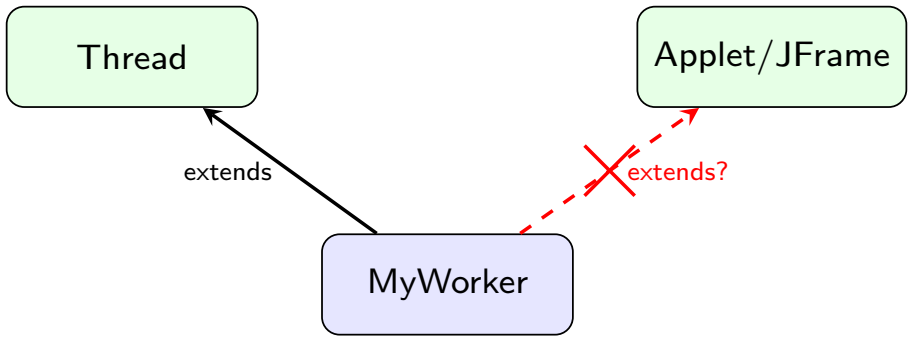


Example: Using sleep() Inside run()

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Tick " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}
```

Limitations of Extending Thread

- Java supports single inheritance: extending `Thread` blocks extending other classes
- Constraint when class logically needs to extend e.g. GUI base class
- Implementing `Runnable` is often preferred for flexibility



Key Thread Class Methods: Quick Reference

Method	Purpose
`start()`	Begins a new thread of execution, which calls `run()`
`run()`	Contains code to execute; overridden by subclass
`sleep(ms)`	Static method pausing current thread for $\approx$ `ms` milliseconds
`getName()` / `setName()`	Get or assign the thread's name
`getId()`	Returns the thread's unique JVM-assigned identifier
`currentThread()`	Static method returning reference to currently executing thread

Key Takeaways

- Create thread by extending `Thread`, overriding `run()`, calling `start()`
- Multiple `Thread` objects can run concurrently with unpredictable interleaving
- `getName()`, `setName()`, `getId()`, `currentThread()` let code identify threads
- `Thread.sleep(ms)` pauses thread, requires handling checked `InterruptedException`

Next Lecture Preview

- Lecture 36 covers the second approach: implementing the `Runnable` interface
- Closes Unit 4 with exception handling and multithreading wrap-up

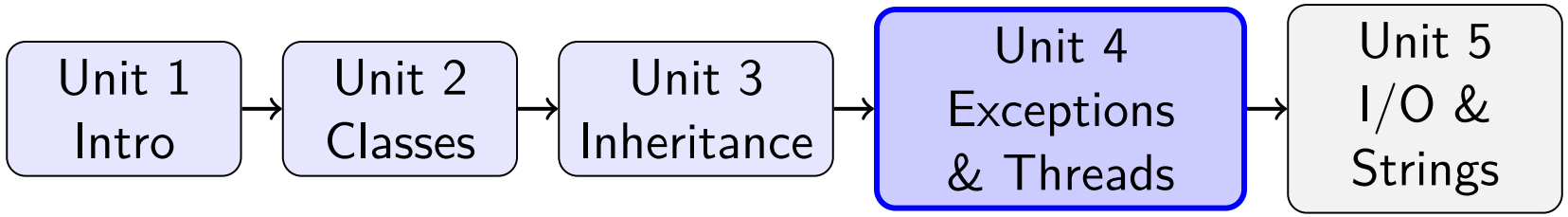
Questions?

Recap: Extending Thread

- Created by extending `Thread`, overriding `run()`, calling `start()`
- `getName()`/`setName()`, `getId()`, and `currentThread()` for thread identity
- `Thread.sleep(ms)` pauses execution (requires catching `InterruptedException`)

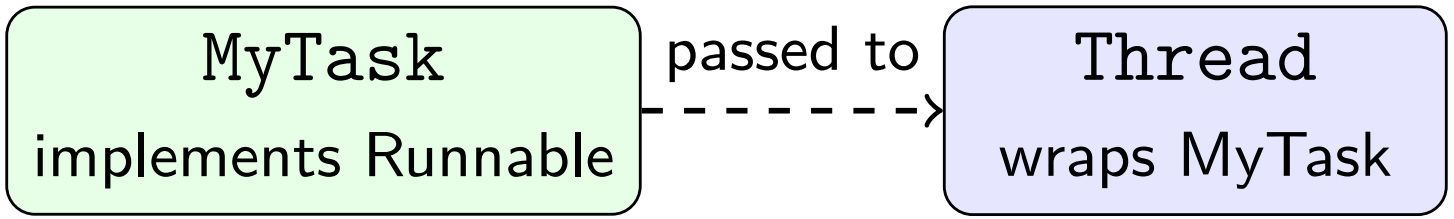
Course Overview & Today's Agenda

- Implementing `Runnable`: Steps and example
- Wrapping a `Runnable` in a `Thread` and starting it
- `Runnable` vs extending `Thread`: Practical scenario
- Unit 4 wrap-up: Exceptions & multithreading



Steps to Implement the Runnable Interface

- **1.** Define class implementing `java.lang.Runnable`
- **2.** Implement single abstract method `run()`
- **3.** Create instance of class (plain task object)
- **4.** Wrap in Thread: `new Thread(taskObject)`
- **5.** Call `start()` to begin execution



Example: A Basic Runnable Task

```
class MyTask implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task count: " + i);
        }
    }
}
```

Example: Wrapping and Starting the Runnable

```
public class RunnableDemo {
    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread t = new Thread(task);
        t.start();
        System.out.println("Main thread continues...");
    }
}
// Thread constructor: Thread(Runnable target) stores the task
    and runs it on start()
```

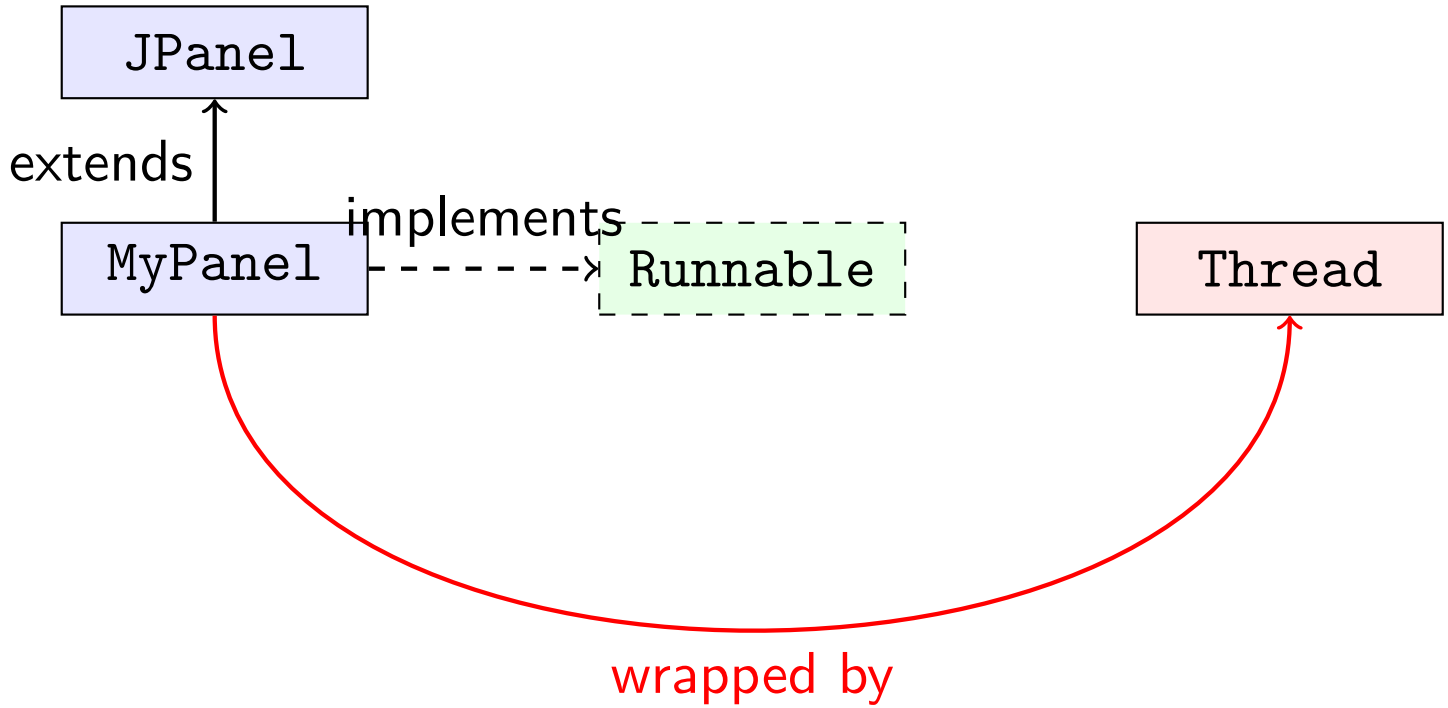
`t.start()` internally calls `task.run()`

Anonymous Class and Lambda Runnable (Java 8+)

```
// Anonymous class form:
Thread t1 = new Thread(new Runnable() {
    public void run() {
        System.out.println("Running via anonymous Runnable");
    }
});
t1.start();

// Lambda form, since Runnable has exactly one abstract method:
Thread t2 = new Thread(() -> System.out.println("Running via
    lambda Runnable"));
t2.start();
```

Practical Scenario: Runnable vs Extending Thread

- Suppose `MyPanel` extends `JPanel` (GUI component base class)
- It **CANNOT** also extend `Thread` (single inheritance rule)
- By implementing `Runnable`, `MyPanel` defines `run()` and keeps `JPanel` parent



Best Practice: Prefer Runnable

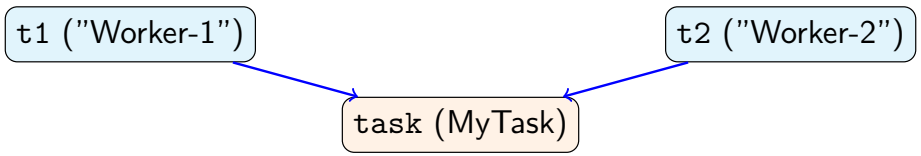
- Prefer `Runnable` when class may need to extend another class in future
- Single `Runnable` can be passed to multiple `Thread` objects (sharing task logic)
- Extending `Thread` is acceptable for simple, standalone thread classes without other inheritance needs

Guideline

Choose **implements Runnable** for flexibility and sharing.
Choose **extends Thread** for quick, standalone thread classes.

Example: Shared Task Across Threads

```
class MyTask implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName() + "
            running");
    }
}
public class SharedTaskDemo {
    public static void main(String[] args) {
        MyTask task = new MyTask();              // one task object
        Thread t1 = new Thread(task, "Worker-1");
        Thread t2 = new Thread(task, "Worker-2");
        t1.start();
        t2.start();
    }
}
```



Final Comparison: Thread vs Runnable

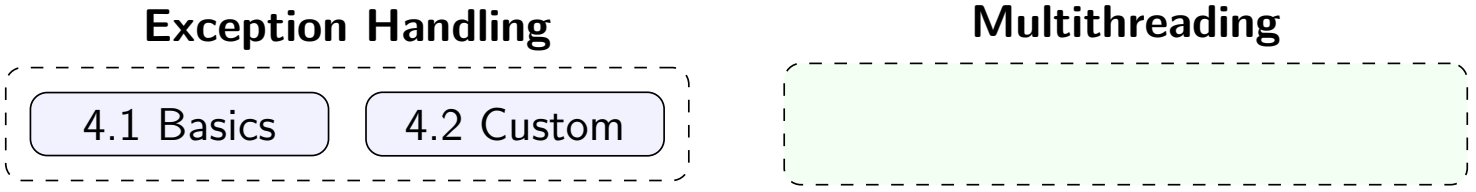
`extends Thread`	`implements Runnable`
Class **IS-A** Thread	Class **HAS-A** task
Cannot extend anything else	Free to extend another class
Simple for standalone thread classes	Supports sharing one task across multiple threads

Common Requirement

Both require defining behavior in `run()` and calling `start()` via a `Thread` object to launch. Never call `run()` directly!

Unit 4 Wrap-Up: Exceptions & Multithreading

- **4.1 Exception Handling:** Hierarchy, `try/catch/finally`, `throw/throws`
- **4.2 Custom Exceptions:** Extending `Exception`, best practices
- **4.3 Thread Concepts:** Process vs thread, lifecycle (5 states)
- **4.4 Thread Creation:** Extending `Thread` vs implementing `Runnable`

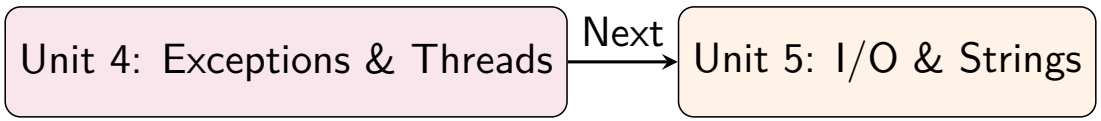


Key Takeaways

- Class implementing `Runnable` defines task in `run()`, wrapped in `Thread`, started with `start()`
- `Runnable` keeps class free to extend another class (avoids using single-inheritance slot)
- Single `Runnable` instance can be shared across multiple `Thread` objects
- Unit 4 covered exception handling (try/catch, hierarchy) and multithreading (lifecycle, creation)

Next Lecture Preview

- Lecture 37 opens Unit 5 with the `String` class: text representation, key methods, and **string immutability**.



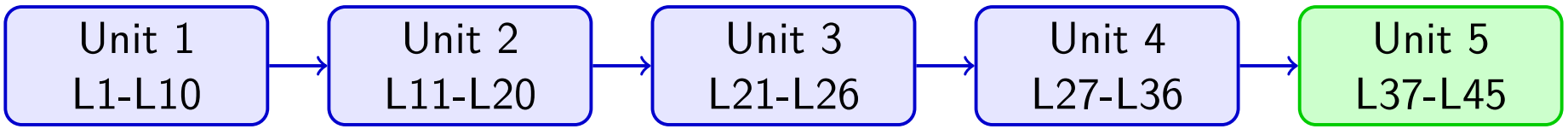
Questions?

Recap: Closing Unit 4 - Threads via Runnable

- Lecture 36 covered creating threads by implementing the Runnable interface, an alternative to extending Thread
- It also reviewed all of Unit 4: exception hierarchy and try-catch-finally, custom exceptions, multithreading concepts, and thread creation approaches
- Today we begin Unit 5, moving from control-flow and concurrency to Java's core data-handling tools, starting with String

Course Overview & Today's Agenda

- Unit 5 roadmap: Strings (L37-38), Collections (L39-41), File Handling (L42-43), Serialization (L44-45)
- Part 1: The String class - creation, storage, and core methods
- Part 2: String immutability - what it means and why it matters



The String Class: What Is a String in Java?

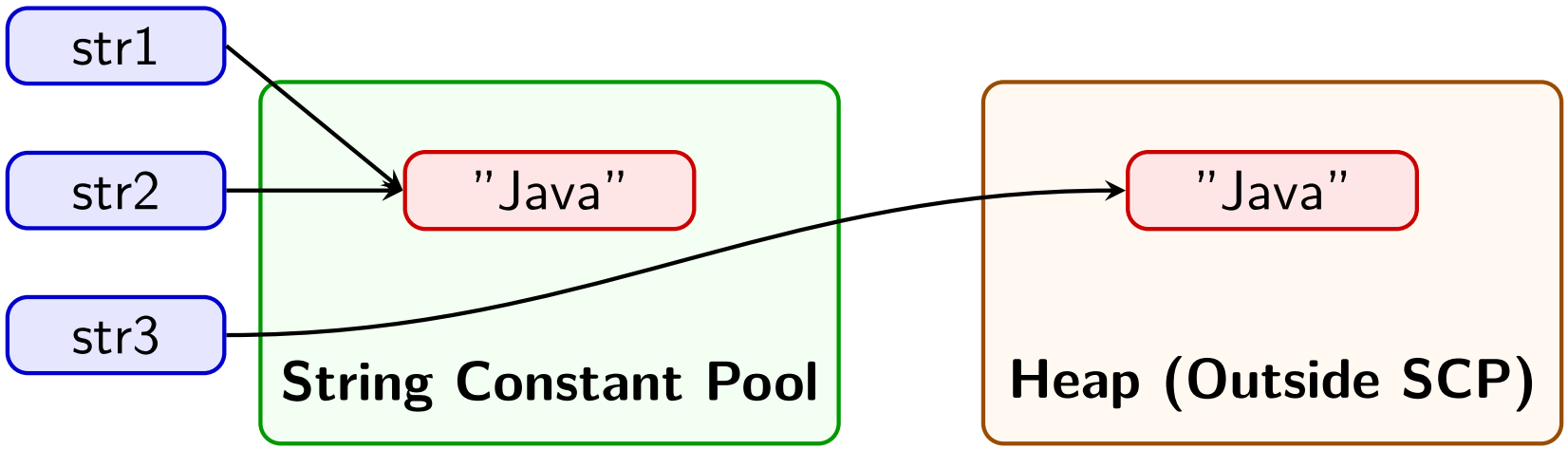
Key Concept

A String in Java is an object of the `java.lang.String` class, not a primitive type.

- Strings represent sequences of characters and are heavily used
- String literals are automatically created as String objects:
  `String name = "Asha";`
- Strings can also be created explicitly with `new`:
  `String name2 = new String("Asha");`

The String Constant Pool

- `String str1 = "Java";` → stored in String Constant Pool (SCP)
- `String str2 = "Java";` → reuses SAME object from SCP
- `String str3 = new String("Java");` → forces NEW heap object



Core String Methods - Part 1

```
String s = "Diploma Engineering";
int len = s.length();                    // 20
char c = s.charAt(8);                    // 'E'
String sub = s.substring(8);             // "Engineering"
String sub2 = s.substring(0, 7);         // "Diploma"
int idx = s.indexOf("Eng");              // 8
```

Core String Methods - Part 2

```
String s = "  Java Programming  ";
String trimmed = s.trim();                    // "Java Programming"
String upper = trimmed.toUpperCase();         // "JAVA PROGRAMMING"
String lower = trimmed.toLowerCase();         // "java programming"
boolean has = trimmed.contains("Prog");       // true
String replaced = trimmed.replace("Java", "C++"); // "C++
    Programming"
```

String Methods Reference Table

Method	Purpose
`length()`	Returns number of characters
`charAt(int i)`	Returns character at index `i`
`substring(int begin, int end)`	Extracts a portion of the string
`indexOf(String s)` / `lastIndexOf(String s)`	Finds position of a substring
`equals(Object o)` / `equalsIgnoreCase(String s)`	Compares content
`compareTo(String s)`	Lexicographic comparison, returns `int`
`toCharArray()`	Converts String to a `char[]` array
`isEmpty()` / `isBlank()`	Checks for zero-length or whitespace-only

== vs equals(): Comparing Strings Correctly

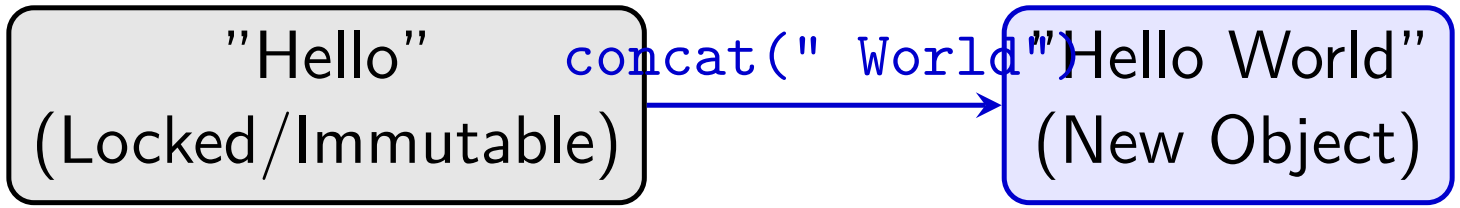
```
String a = "hello";
String b = "hello";
String c = new String("hello");

System.out.println(a == b);      // true  - both point to the
    same pooled literal
System.out.println(a == c);      // false - c is a distinct heap
     object
System.out.println(a.equals(c)); // true  - equals() compares
    character content
```

Operator/Method	Compares	Best For
==	Memory Reference (Address)	Checking object identity
equals()	Character Content (Value)	Normal string comparison

String Immutability: What It Means

- Once a String object is created, its character content can never be changed
- Every String method that appears to 'modify' a string actually returns a brand-new String object

```
String s = "Hello";
s.concat(" World");  // s is STILL "Hello" - result discarded!
String s2 = s.concat(" World"); // s2 is "Hello World"; s
    unaffected
```



Why Is String Designed to Be Immutable?

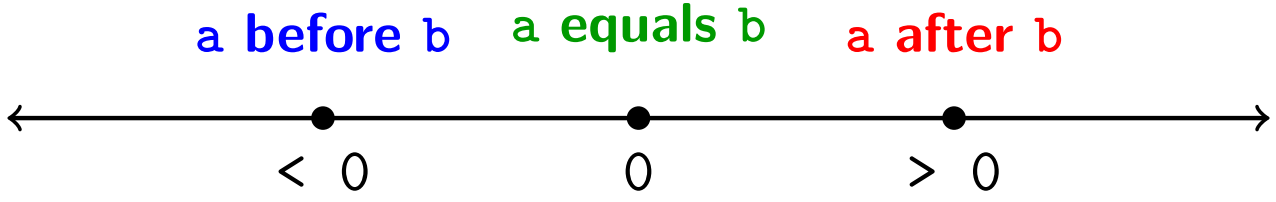
- **Security:** strings are used for file paths, network connections, and class names; immutability prevents them from being altered after being validated
- **String pool sharing:** safe sharing of literals is only possible because shared objects cannot be silently changed
- **Thread safety:** an immutable object can be freely shared across multiple threads without synchronization
- **Hashcode caching:** String's `hashCode()` can be computed once and cached, speeding up use as `HashMap`/`HashSet` keys

Immutability in Action: A Common Pitfall

```
public class ImmutableDemo {
    public static void main(String[] args) {
        String greeting = "Hi";

        greeting.concat(", there!");    // BUG: return value
            discarded
        System.out.println(greeting);  // prints "Hi", NOT "Hi,
            there!"

        greeting = greeting.concat(", there!"); // correct:
            reassign
        System.out.println(greeting);  // prints "Hi, there!"
    }
}
```

String Comparison and Ordering with compareTo()

```
String a = "apple";
String b = "banana";
int result = a.compareTo(b);
// result < 0 because 'apple' comes before 'banana'
    lexicographically

if (result < 0) { System.out.println(a + " comes first"); }
else if (result > 0) { System.out.println(b + " comes first"); }
else { System.out.println("Equal strings"); }
```



Key Takeaways

- **String is a class:** literals are pooled in the String Constant Pool while `new String(...)` forces a separate heap object
- **Comparison:** Use `equals()` to compare String content and == only to compare references
- **Immutability:** String objects are immutable; every 'modifying' method returns a new String rather than changing the original
- **Benefits:** Immutability gives String its security, pool-sharing, thread-safety, and hashcode caching

Next Lecture Preview

Coming Up: Lecture 38

Lecture 38 builds on today's foundation by introducing `StringBuilder` and `StringBuffer` as mutable alternatives to `String` for heavy text-building work, plus regular expressions for pattern-based string operations.

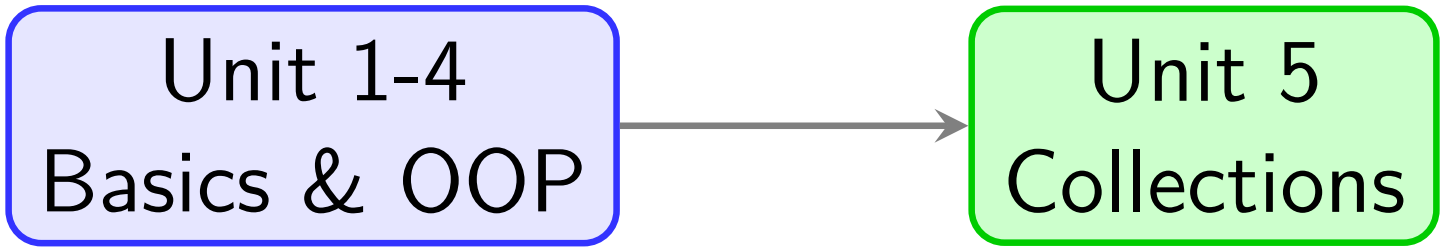
Questions?

Recap: String Class and Immutability

- Lecture 37 covered String creation, the constant pool, and core methods like `substring()`, `indexOf()`, and `compareTo()`
- String is immutable - `concat()`, `replace()` always return a new object, never modify the original
- Immutability is safe but costly when a string must be modified many times

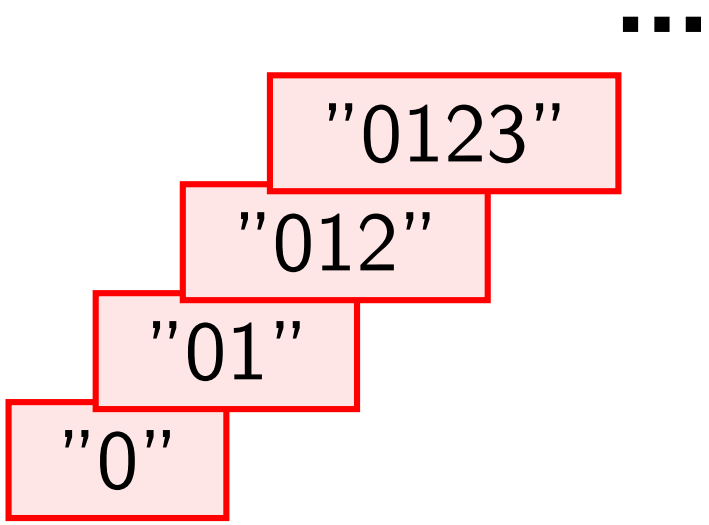
Course Overview & Today's Agenda

- Part 1: **StringBuilder** - a mutable companion to String
- Part 2: **StringBuffer** and thread-safe string building
- Part 3: **Regular expressions** for pattern-based string operations



The Problem: String Concatenation in a Loop

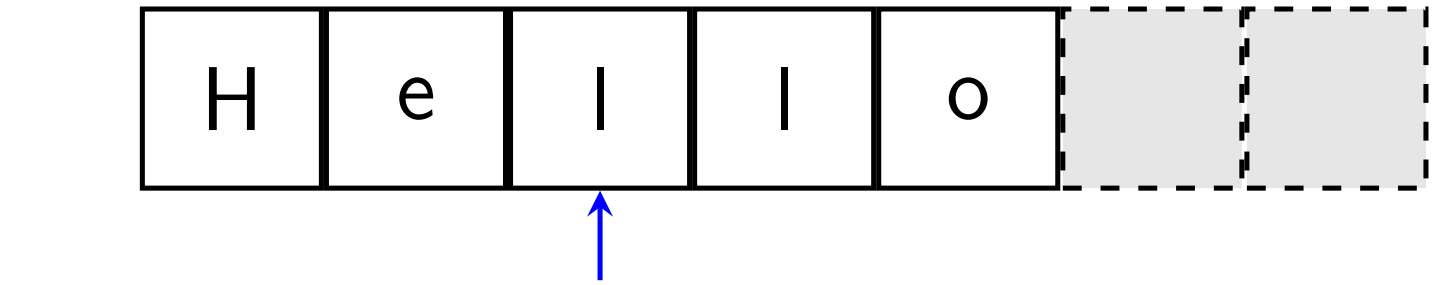
```
String result = "";
for (int i = 0; i < 10000; i++) {
    result = result + i;    // creates a NEW String object every
        single iteration
}
// 10,000 iterations -> up to 10,000 discarded intermediate
    String objects
```



Garbage memory accumulation!

StringBuilder: A Mutable Sequence of Characters

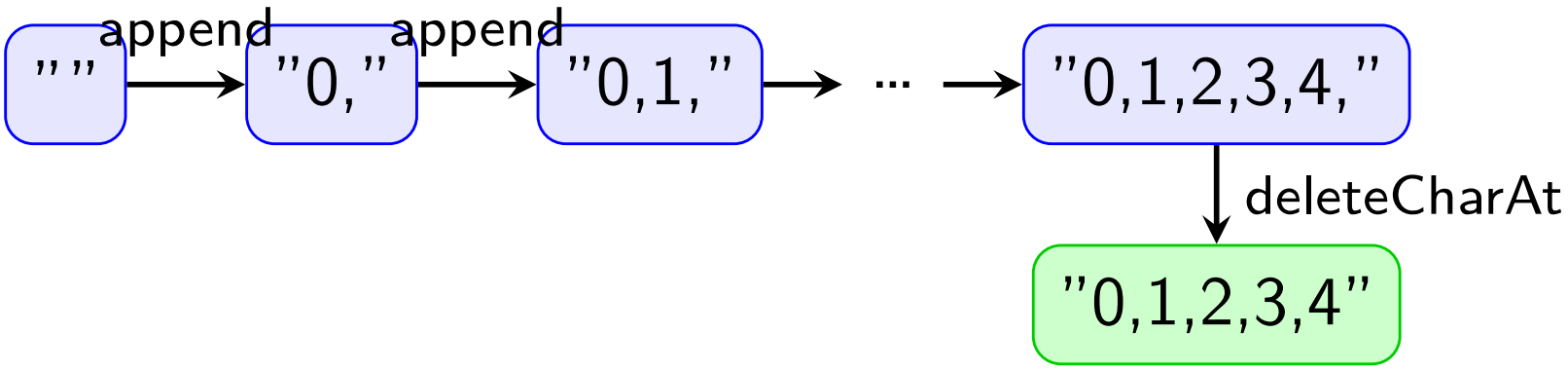
- **StringBuilder** stores characters in a resizable, mutable internal buffer (no new object on change)
- Usage: `StringBuilder sb = new StringBuilder();`
- Core methods: `append()`, `insert()`, `delete()`, `reverse()`
- Converting back: `String finalStr = sb.toString();`



Internal Capacity $>$ Length (Expandable)

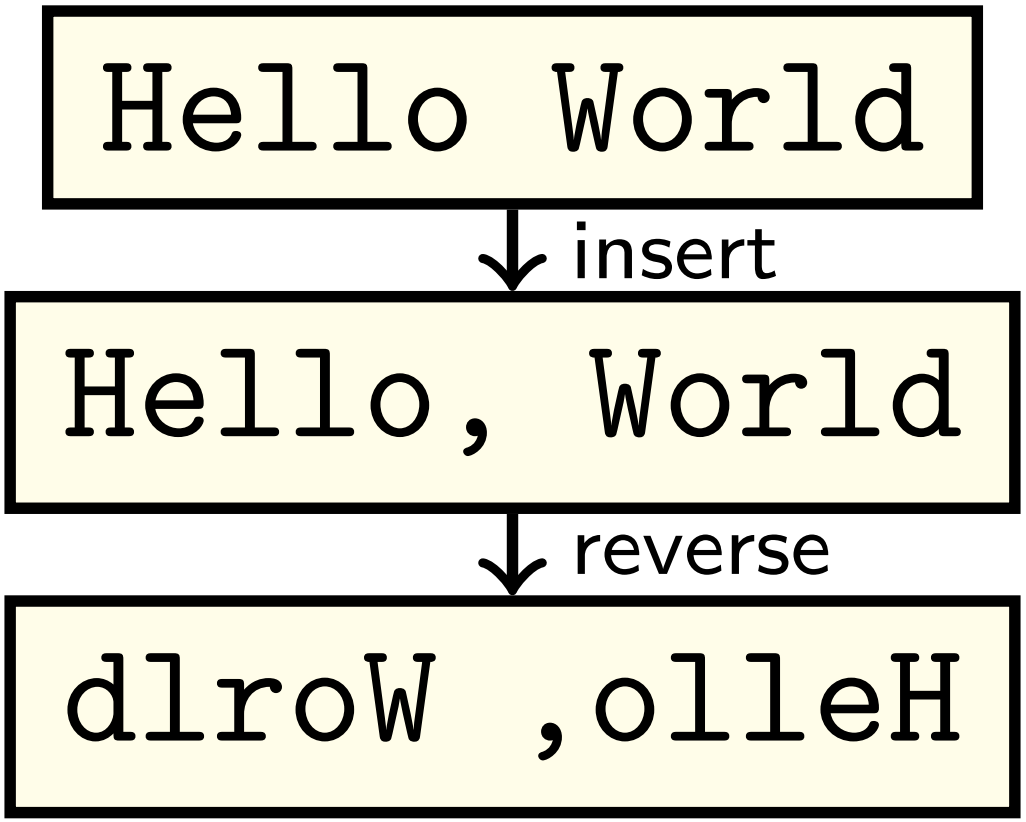
StringBuilder in Action

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
    sb.append(i).append(",");    // method chaining
}
sb.deleteCharAt(sb.length() - 1);    // remove trailing comma
System.out.println(sb.toString());   // prints: 0,1,2,3,4
```



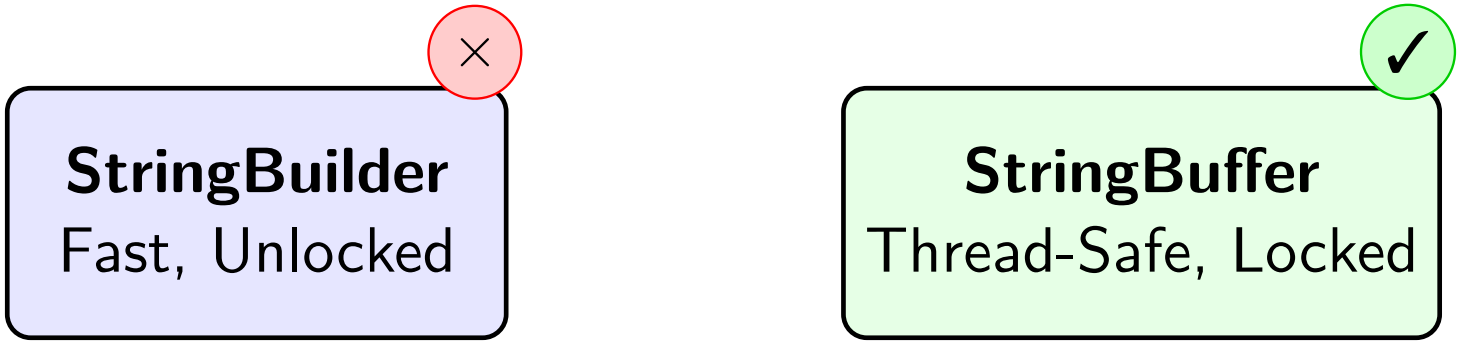
More StringBuilder Methods

```
StringBuilder sb = new StringBuilder("Hello World");
sb.insert(5, ",");                  // "Hello, World"
sb.reverse();                        // "dlroW ,olleH"
sb.reverse();                        // back to "Hello, World"
sb.replace(0, 5, "Hi");              // "Hi, World"
System.out.println(sb);              // "Hi, World"
```



StringBuffer: The Thread-Safe Alternative

- **StringBuffer** has almost the identical API to `StringBuilder`
- Key difference: Methods are **synchronized**, safe for concurrent modification
- Cost of safety: Synchronization adds overhead, slower in single-threaded code



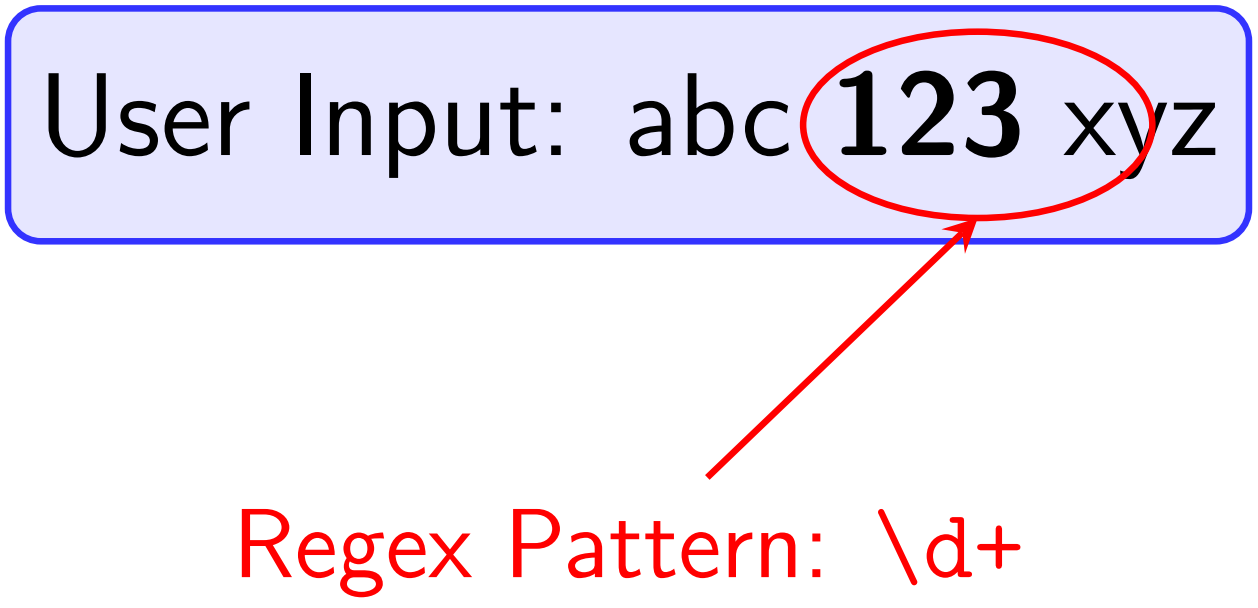
String vs StringBuilder vs StringBuffer

- **Rule of thumb:** String for fixed text, StringBuilder for single-threaded building, StringBuffer for multi-threaded building

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Safe (by design)	Not Synchronized	Synchronized (Safe)
Performance	Slowest (modifying)	Fastest	Slower than Builder
Best Use Case	Fixed text	Single-threaded	Multi-threaded

Introducing Regular Expressions in Java

- Regular expression (regex): a pattern describing a set of matching strings
- E.g., `\d+` matches one or more digits
- Java supports regex via `String` methods & `Pattern/Matcher` classes
- Use cases: Input validation (emails, phone numbers), extracting text



Regex with String Methods

```
String phone = "9876543210";
boolean valid = phone.matches("\\d{10}");   // true - exactly 10
      digits

String sentence = "Java is fun, Java is powerful";
String noJava = sentence.replaceAll("Java", "Python"); //
     replace ALL matches

String csv = "Asha,Ravi,Meera";
String[] names = csv.split(",");   // {"Asha", "Ravi", "Meera"}
```

Pattern and Matcher: Reusable Regex Objects

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

Pattern p = Pattern.compile("[a-zA-Z]+");
Matcher m = p.matcher("Room 101B has 3 chairs");
while (m.find()) {
    System.out.println(m.group()); // prints: Room, then B, then
        has, then chairs
}
```

Common Pitfalls with Strings, Builders, and Regex

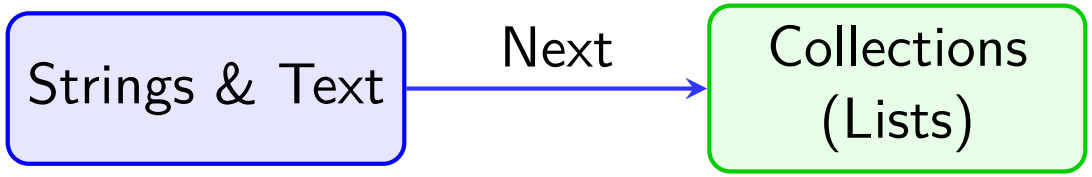
- **Immutability Bug:** Forgetting to reassign String method results. (StringBuilder mutates in place, no reassignment needed)
- **Defaulting to StringBuffer:** Wastes performance if no multithreading is involved; prefer StringBuilder
- **Escaping Backslashes:** Regex `\d` must be written as `\\d` in source code
- **Redundant toString():** Calling `toString()` on a StringBuilder when `println()` already calls it implicitly

Key Takeaways

- **StringBuilder:** Mutable, resizable character buffer. Avoids object-churn of repeated String concatenation.
- **StringBuffer:** Same API as StringBuilder but with synchronized methods for thread safety (slower).
- **Rule of Thumb:** String for fixed text, StringBuilder for single-threaded, StringBuffer for multi-threaded.
- **Regex:** `String` methods (`matches`/`replaceAll`/`split`) for quick use, `Pattern`/`Matcher` for reuse.

Next Lecture Preview

- Lecture 39 shifts from individual strings to collections of data
- Opening Unit 5's Collections Framework topic with the `List` interface and `ArrayList` & `LinkedList`



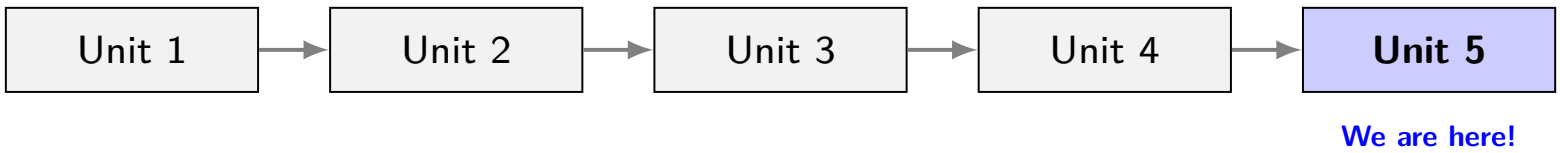
Questions?

Recap: StringBuilder, StringBuffer, and Regex

- Lecture 38 introduced `StringBuilder` for efficient text building and `StringBuffer` for thread-safe building.
- We compared `String` vs `StringBuilder` vs `StringBuffer` (mutability, thread-safety, performance).
- Introduced basic regular expressions via `matches()`, `replaceAll()`, `split()`, and `Pattern`/`Matcher`.

Course Overview & Today's Agenda

- **Part 1**: Why Collections Framework exists (array limits).
- **Part 2**: Hierarchy (`Collection`, `List`, `Set`, `Map`).
- **Part 3**: The `List` interface (`ArrayList` and `LinkedList`).

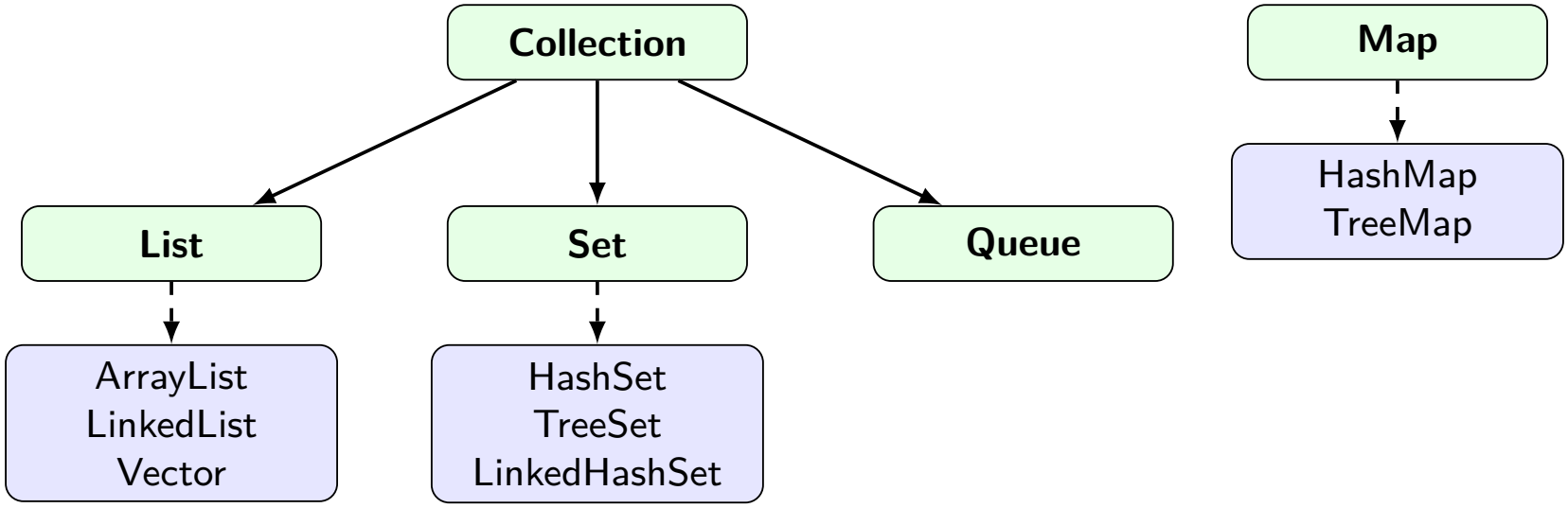


The Problem with Plain Arrays

- Arrays have a **FIXED** size once created: `int[] arr = new int[5];` cannot grow to hold a 6th element.
- Arrays provide no built-in methods for searching, sorting, inserting in the middle, or removing an element.
- Arrays can only hold one data type (or `Object`, sacrificing type safety).
- Real-world data (a growing list of students, shopping cart) rarely has a fixed size known in advance.

Rigid Array: [ ][ ][ ] **VS** Flexible Collection: [ ][ ][ ][ ]...

The Java Collections Framework: The Big Picture



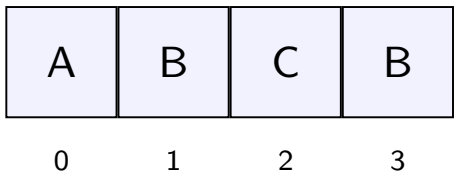
- **Collection**: Root representing a group of objects.
- **Map**: Separate interface for key-value pairs.

The List Interface: Ordered, Allows Duplicates

- A `List` maintains **insertion order** – elements stay in the order they were added, accessible by index.
- Unlike `Set`, a `List` **allows duplicate elements**.
- Declared using the interface type on the left (good practice):

```
List<String> names = new ArrayList<>();
```

- The <> is a **generic type parameter** – it enforces compile-time type safety (e.g., only `String` objects allowed).



Duplicates allowed!

ArrayList: A Resizable Array

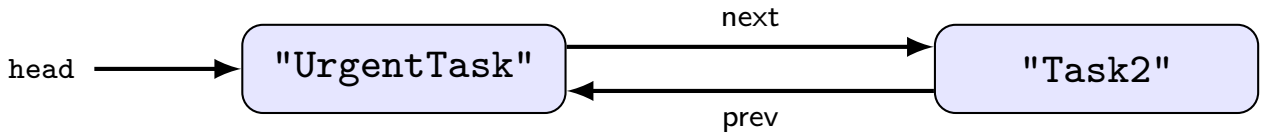
- Backed by a resizable array internally.
- Fast for accessing elements `get(index)`, but inserting/removing in the middle requires shifting elements.

```
List<String> names = new ArrayList<>();
names.add("Asha");
names.add("Ravi");
names.add("Meera");
names.add(1, "Deep");          // insert at index 1, shifts others
    right
System.out.println(names);     // [Asha, Deep, Ravi, Meera]
System.out.println(names.get(2));  // Ravi
```

LinkedList: A Doubly-Linked Chain of Nodes

- Inserting/removing at the front is cheap (no shifting needed).

```
List<String> queue = new LinkedList<>();
queue.add("Task1");
queue.add("Task2");
queue.add(0, "UrgentTask");    // insert at front - cheap for
    LinkedList
queue.remove("Task1");
System.out.println(queue);      // [UrgentTask, Task2]
```



ArrayList vs LinkedList

ArrayList	**LinkedList**
Backed by a resizable array	Backed by a doubly-linked list of nodes
Fast `get(index)` – $\mathcal{O}(1)$	Slow `get(index)` – $\mathcal{O}(n)$ traversal
Slow insert/remove in the middle – $\mathcal{O}(n)$ shift	Fast insert/remove at known position – $\mathcal{O}(1)$
Default choice for most use cases (frequent reads, occasional writes)	Suits frequent insertions/deletions at the ends, e.g. queues or stacks

Common List Methods Reference

Method	Purpose
`add(E e) / add(int index, E e)`	Append or insert an element
`get(int index)`	Retrieve element at a position
`set(int index, E e)`	Replace element at a position
`remove(int index) / remove(Object o)`	Remove by position or by value
`size()`	Number of elements currently stored
`contains(Object o)`	Checks if an element exists
`isEmpty()`	Checks if the list has zero elements
`clear()`	Removes all elements

Iterating a List with the Enhanced for-loop

- The enhanced `for`-loop is the simplest, most readable way to visit every element in order.

```
List<String> fruits = new ArrayList<>();
fruits.add("Mango");
fruits.add("Apple");
fruits.add("Banana");
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

Storing Objects in a List: Beyond Strings

- Collections routinely store custom class instances, connecting directly back to Unit 2 (Object-Oriented fundamentals).

```
class Student {
    String name;
    int rollNo;
    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }
}

List<Student> students = new ArrayList<>();
students.add(new Student("Asha", 1));
students.add(new Student("Ravi", 2));
System.out.println(students.get(0).name);    // Asha
```

Key Takeaways

- The Collections Framework solves the fixed-size and lack of built-in operations limitations of plain arrays.
- **Collection** is the root interface (`List`, `Set`, `Queue`); **Map** is a parallel interface for key-value pairs.
- **List** is an ordered collection that allows duplicates.
- `ArrayList` provides fast random access ($\mathcal{O}(1)$), while `LinkedList` provides fast insertions/deletions at known positions.
- Lists can safely store custom class objects (`Student`, etc.) using Generics (`<T>`).

Next Lecture Preview

- Lecture 40 continues the Collections Framework story with the **Set** interface.
- We will cover `HashSet` and `TreeSet` for storing unique, duplicate-free collections.

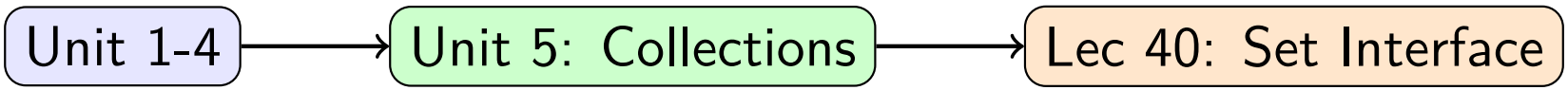
Questions?

Recap: The List Interface

- Lecture 39 introduced the Collections Framework and its Collection/List/Set/Map hierarchy
- It covered the List interface in depth: ordered, allows duplicates, with ArrayList and LinkedList as the two main implementations
- Today we move to the second branch of that hierarchy: the Set interface

Course Overview & Today's Agenda

- Part 1: The Set interface - no duplicates allowed
- Part 2: HashSet - fast, unordered uniqueness
- Part 3: TreeSet - automatically sorted uniqueness

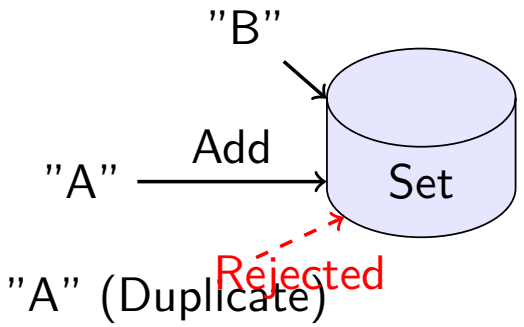


The Set Interface: Uniqueness Guaranteed

What is a Set?

A Set is a collection that contains NO duplicate elements - adding an element that already exists has no effect.

- Unlike List, a Set generally does not guarantee any particular iteration order
- Declared via the interface type: `Set<String> tags = new HashSet<>();`
- Common use cases: storing unique usernames, removing duplicates from a dataset



HashSet: Fast, Unordered Storage

```
Set<String> colors = new HashSet<>();
colors.add("Red");
colors.add("Green");
colors.add("Blue");
colors.add("Red");                  // duplicate - silently ignored
System.out.println(colors.size());  // 3, not 4
System.out.println(colors);          // order NOT guaranteed, e.
    g. [Blue, Red, Green]
```

TreeSet: Automatically Sorted Storage

```
Set<Integer> scores = new TreeSet<>();
scores.add(85);
scores.add(42);
scores.add(99);
scores.add(42);                    // duplicate - ignored
System.out.println(scores);  // [42, 85, 99] - always sorted
    ascending
```

HashSet vs TreeSet

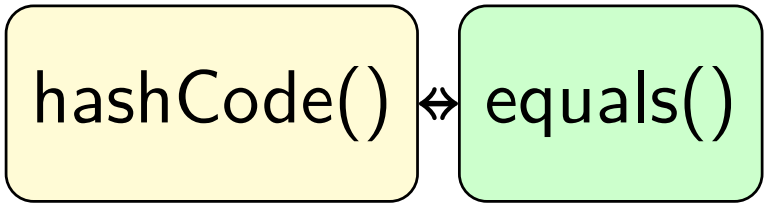
Feature	**HashSet**	**TreeSet**
Backing Data Structure	Hash table	Balanced tree (Red-Black tree)
Order	No guaranteed order	Always sorted ascending
Performance	Fastest $O(1)$ average	Slower $O(\log n)$
Best Use Case	Only uniqueness needed	Uniqueness AND sorted order

The hashCode() and equals() Contract

The Contract

`HashSet` decides if objects are 'the same' using `hashCode()` and `equals()` together. Both must be overridden consistently!

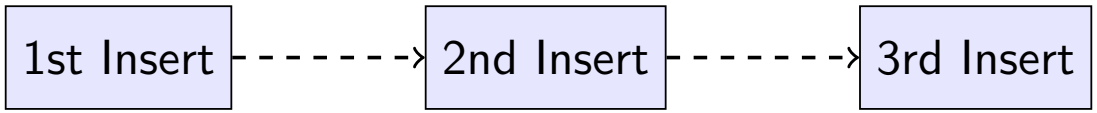
- Both methods are inherited from the `Object` class
- `String` and wrapper classes (`Integer`, `Double`) already override both correctly



Must match consistently!

LinkedHashSet: The Middle Ground

- `LinkedHashSet` combines `HashSet`'s uniqueness with a predictable iteration order: insertion order is preserved
- Slightly more memory overhead than `HashSet` but avoids `TreeSet`'s sorting cost
- Useful when duplicates must be rejected AND original insertion order matters

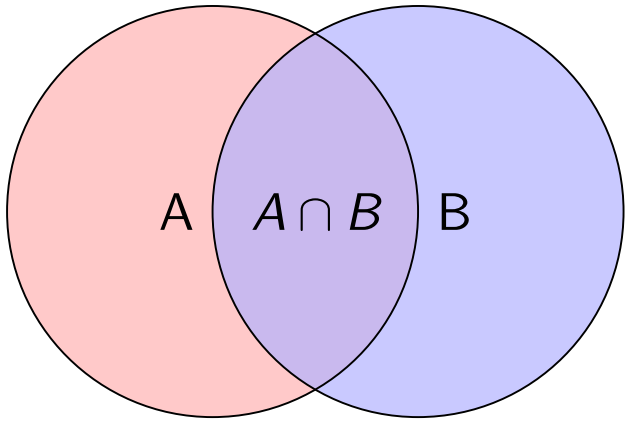


Removing Duplicates from a List Using a Set

```
List<String> withDuplicates = new ArrayList<>();
withDuplicates.add("A"); withDuplicates.add("B"); withDuplicates
    .add("A");
Set<String> unique = new HashSet<>(withDuplicates);    //
    constructor removes duplicates
List<String> cleanList = new ArrayList<>(unique);      // convert
     back to a List if needed
System.out.println(cleanList.size());  // 2
```

Iterating and Basic Set Operations

```
Set<Integer> setA = new HashSet<>(List.of
    (1, 2, 3, 4));
Set<Integer> setB = new HashSet<>(List.of
    (3, 4, 5, 6));

Set<Integer> union = new HashSet<>(setA);
union.addAll(setB);                 // union:
    {1,2,3,4,5,6}

Set<Integer> intersection = new HashSet
    <>(setA);
intersection.retainAll(setB);  //
    intersection: {3,4}

for (int n : union) { System.out.print(n
    + " "); }
```



Common Set Pitfalls

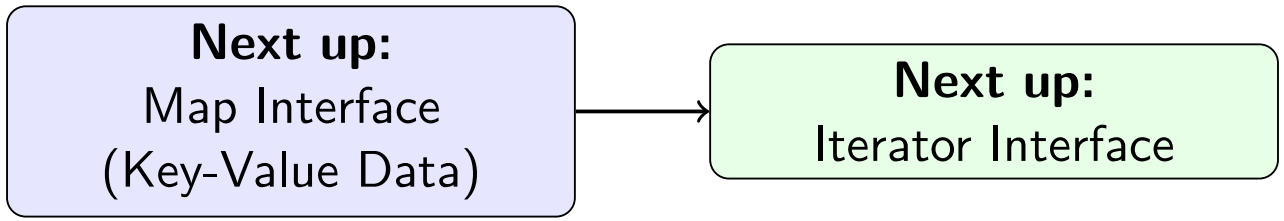
- Expecting `HashSet` iteration to follow insertion order - it does not; use `LinkedHashSet` if order matters
- Storing mutable custom objects in a `HashSet` and changing a field used in `hashCode()` after insertion, making it 'unfindable'
- Forgetting that `TreeSet` requires elements to be mutually comparable (via `Comparable` or a `Comparator`)
- Assuming `add()` on a Set always increases `size()` - always check the boolean return value

Key Takeaways

- `Set` is a collection that guarantees no duplicate elements
- `HashSet` gives fastest average performance but no guaranteed iteration order
- `TreeSet` keeps elements always sorted at the cost of speed
- `LinkedHashSet` offers uniqueness plus predictable insertion-order iteration
- Correct `HashSet` behaviour for custom objects requires overriding `hashCode()` and `equals()`

Next Lecture Preview

- Lecture 41 completes the Collections Framework tour with the Map interface (`HashMap` and `TreeMap`) for key-value data, plus a formal look at the `Iterator` interface and the enhanced for-loop.



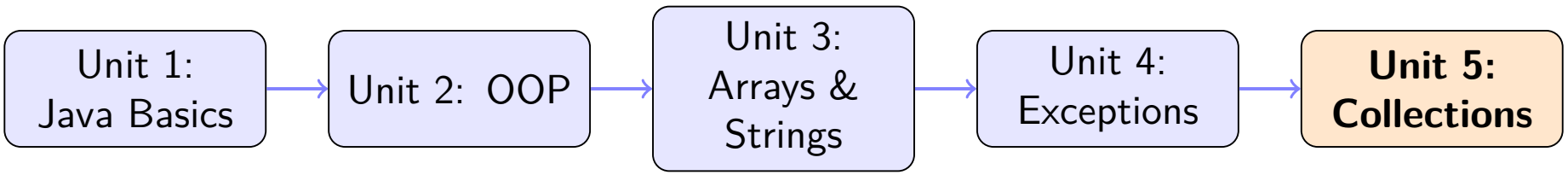
Questions?

Recap: The Set Interface

- Lecture 40 covered Set's core guarantee - no duplicate elements - and its two main implementations
- HashSet offers fast, unordered storage; TreeSet keeps elements automatically sorted using natural ordering
- It also revisited the hashCode()/equals() contract inherited from the Object class, which governs how uniqueness is detected

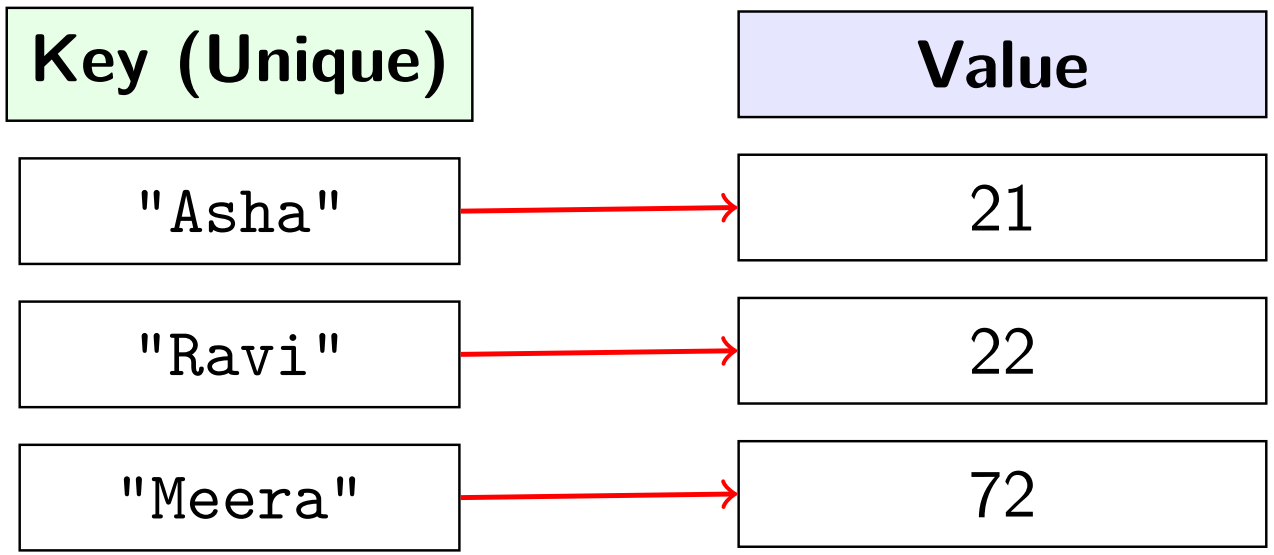
Course Overview & Today's Agenda

- Part 1: The Map interface - storing key-value pairs
- Part 2: HashMap and TreeMap implementations
- Part 3: The Iterator interface and the enhanced for-loop, formalised



The Map Interface: Key-Value Pairs

- A Map stores data as key-value pairs; each key maps to exactly one value, and keys must be unique
- Map is NOT part of the Collection interface hierarchy - it is a separate, parallel interface
- Real-world analogy: dictionary (word $\rightarrow$ meaning), phone book, or student records



HashMap: Fast, Unordered Key-Value Storage

Key Overwriting

When `put()` is used with an existing key, it overwrites the previous value. Keys remain unique.

```
Map<String, Integer> ages = new HashMap<>();
ages.put("Asha", 20);
ages.put("Ravi", 22);
ages.put("Asha", 21);       // same key - overwrites the previous
    value
System.out.println(ages.get("Asha"));    // 21
System.out.println(ages.containsKey("Ravi"));  // true
System.out.println(ages.size());    // 2, not 3
```

TreeMap: Automatically Key-Sorted Storage

Natural Ordering

TreeMap automatically sorts elements by their keys in ascending order, just like a TreeSet.

```
Map<String, Integer> scores = new TreeMap<>();
scores.put("Ravi", 88);
scores.put("Asha", 95);
scores.put("Meera", 72);
System.out.println(scores);  // {Asha=95, Meera=72, Ravi=88} -
    keys sorted alphabetically
```

HashMap vs TreeMap

HashMap	**TreeMap**
Backed by a hash table	Backed by a balanced tree
No guaranteed key order	Keys always sorted ascending
Fastest put/get/remove - $\mathcal{O}(1)$ average	Slower put/get/remove - $\mathcal{O}(\log n)$
Default, general-purpose choice for key-value storage	Used when sorted-key iteration is required (e.g. leaderboard)

Common Map Methods Reference

Method	Purpose
`put(K key, V value)`	Insert or update a key-value pair
`get(K key)`	Retrieve the value for a key (returns `null` if absent)
`containsKey(K key)`	Check for key membership
`containsValue(V value)`	Check for value membership
`remove(K key)`	Deletes the entry for a key
`keySet()`	Returns a `Set` view of all keys
`values()`	Returns a `Collection` view of all values
`entrySet()`	Returns a `Set` of `Map.Entry` objects, for iterating key and value
`getOrDefault(K key, V def)`	Safe retrieval with a fallback value

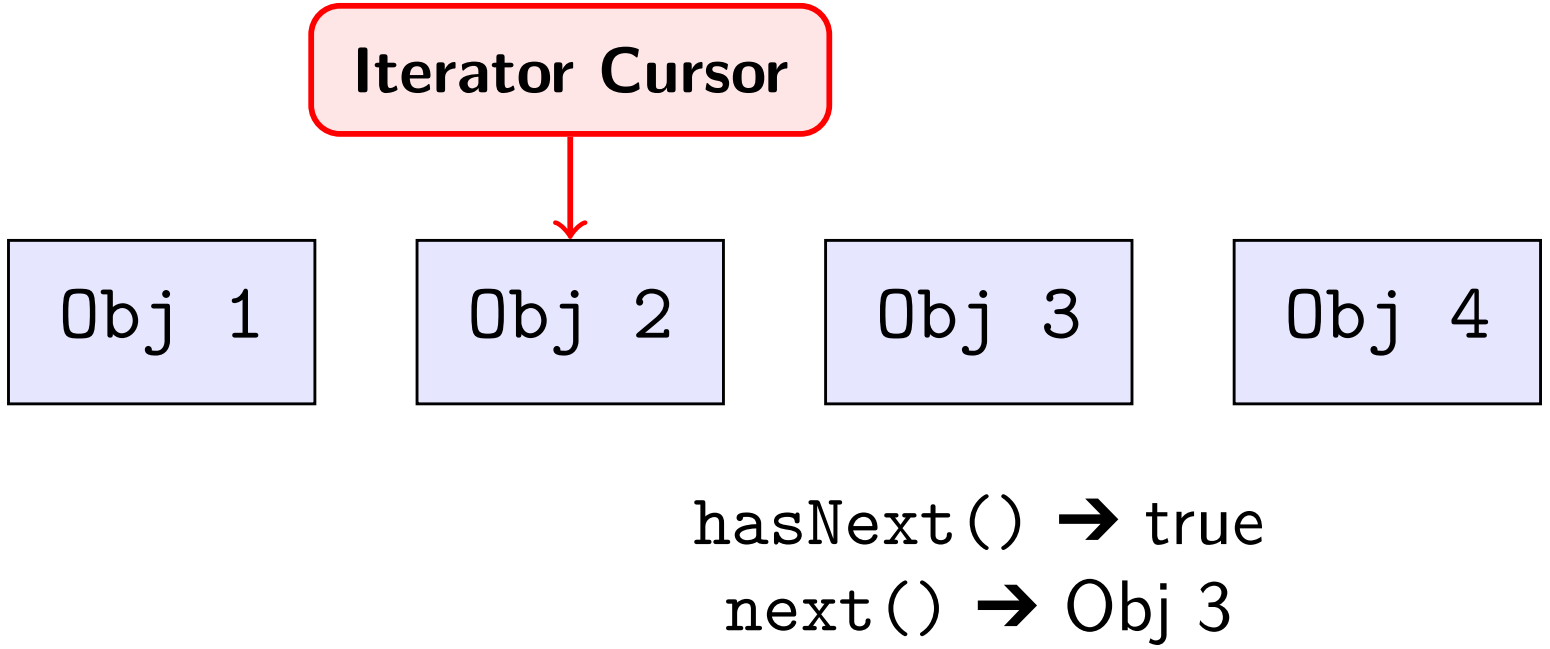
Iterating a Map with entrySet() and Map.Entry

- `entrySet()` provides the most efficient way to access both the key and value simultaneously.

```
Map<String, Integer> ages = new HashMap<>();
ages.put("Asha", 20);
ages.put("Ravi", 22);

for (Map.Entry<String, Integer> entry : ages.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue
        ());
}
```

Formalising Iteration: The Iterator Interface

- Iterator provides a uniform way to step through any collection's elements
- Three core methods: `hasNext()`, `next()`, `remove()`
- The enhanced for-loop is actually syntactic sugar for Iterator calls



Using Iterator Explicitly

- Use explicit Iterator when you need to safely remove elements mid-iteration.

```
List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5,
    6));
Iterator<Integer> it = numbers.iterator();
while (it.hasNext()) {
    int n = it.next();
    if (n % 2 == 0) {
        it.remove();      // safely removes the CURRENT element
            during iteration
    }
}
System.out.println(numbers);  // [1, 3, 5]
```

Enhanced for-loop vs Iterator

Enhanced for-loop	**Iterator**
Concise, readable	More verbose
Ideal for simply reading/processing every element in order	The ONLY safe way to remove elements while iterating
Throws `ConcurrentModificationException` if `remove()` is called directly	Supports safe removal via `it.remove()`
Use by default	Switch to explicit Iterator when removing elements

Key Takeaways

- Map stores unique keys mapped to values and sits outside the Collection interface hierarchy, unlike List and Set
- HashMap offers fast, unordered key-value storage; TreeMap keeps keys automatically sorted, mirroring the HashSet/TreeSet trade-off from Lecture 40
- `entrySet()` with `Map.Entry` is the standard idiom for iterating both keys and values of a Map together
- The Iterator interface (`hasNext`/`next`/`remove`) underlies the enhanced for-loop used since Lecture 39, and is the only safe way to remove elements during iteration, avoiding `ConcurrentModificationException`

Next Lecture Preview

- Lecture 42 leaves in-memory Collections behind and opens Unit 5's file-handling topic, starting with file handling fundamentals and the File class for representing and inspecting files and directories.

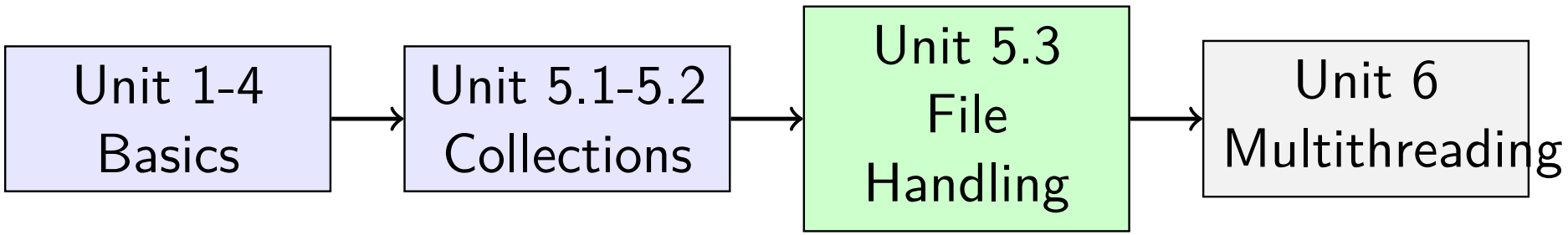
Questions?

Recap: Map Interface and Iteration

- Lecture 41 covered Map interface with HashMap (fast, unordered) & TreeMap (sorted)
- Introduced entrySet() and Map.Entry for iterating keys and values together
- Formalised Iterator interface; enhanced for-loop is syntactic sugar

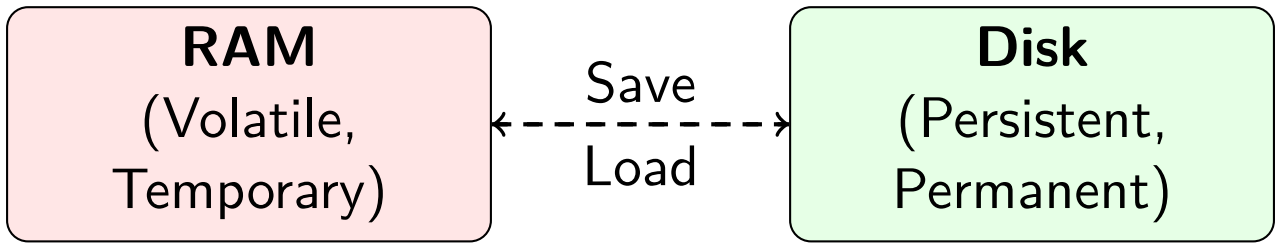
Course Overview & Today's Agenda

- Part 1: Why file handling matters - persistent storage beyond program memory
- Part 2: The java.io package overview
- Part 3: The File class - representing and inspecting files and directories



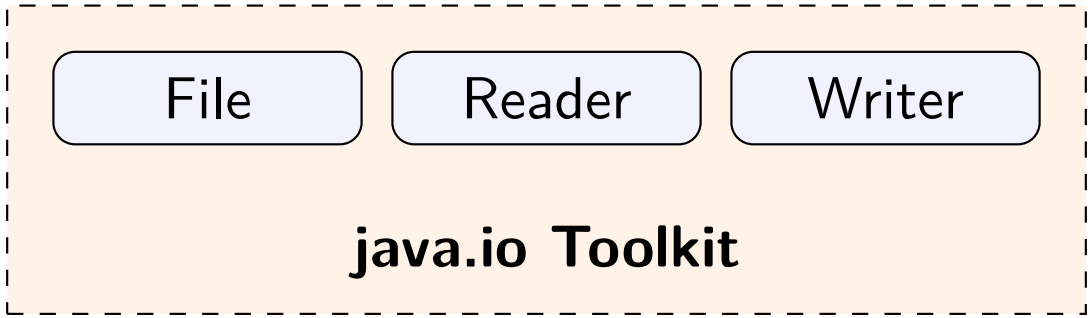
Why File Handling? The Limits of In-Memory Data

- Collections (Lectures 39-41) live in RAM - lost when program terminates
- Real applications need PERSISTENT storage: survives restarts, reboots, and shares
- File handling lets Java read/write to disk, providing persistence
- Common uses: configs, logging, student records



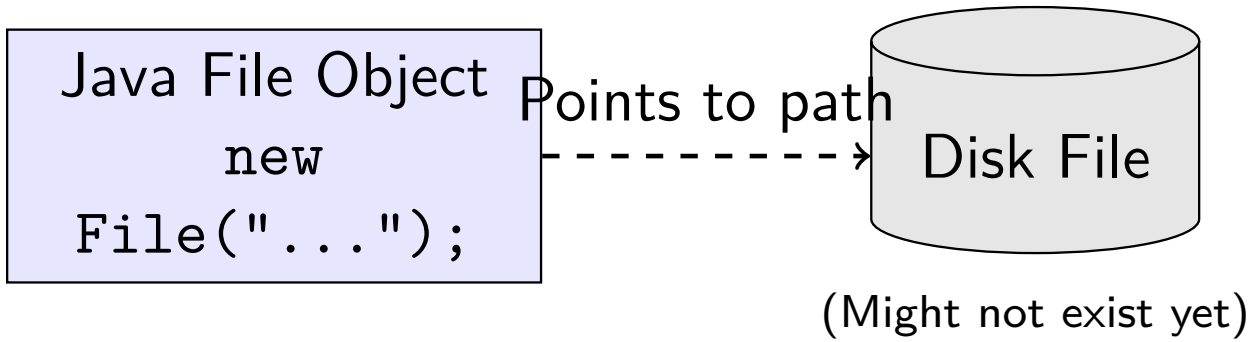
The java.io Package: Java's File-Handling Toolkit

- `java.io` is Java's core package for I/O operations and file handling
- Key classes: File, FileReader/FileWriter, BufferedReader/BufferedWriter
- Most `java.io` operations throw `IOException` (CHECKED exception)
- Requires: `import java.io.*;` or `import java.io.File;`



The File Class: Representing Files and Directories

- A `File` object represents the PATH to a file or directory
- It does NOT automatically create or open the actual file
- Creation: `File f = new File("student.txt");`
- Can represent existing, non-existing, or directory paths
- Content reading/writing requires separate stream classes (Lecture 43)



Creating and Checking Files

```
import java.io.File;
import java.io.IOException;

File f = new File("student.txt");
try {
    boolean created = f.createNewFile();  // true if a new file
        was created
    System.out.println(created ? "File created" : "File already
        exists");
} catch (IOException e) {
    System.out.println("Error creating file: " + e.getMessage())
        ;
}
```

File Class: Common Methods Reference

Method	Purpose
`exists()`	Checks whether the file/directory already exists
`createNewFile()`	Creates a new, empty file (throws `IOException`)
`delete()`	Deletes the file or directory
`length()`	Returns file size in bytes
`getName()` / `getPath()`	Retrieve name and path information
`isFile()` / `isDirectory()`	Checks the type of the path
`mkdir()` / `mkdirs()`	Creates a directory (mkdirs creates missing parents)
`list()` / `listFiles()`	Returns the contents of a directory

Inspecting File Properties

```
File f = new File("student.txt");
if (f.exists()) {
    System.out.println("Name: " + f.getName());
    System.out.println("Path: " + f.getAbsolutePath());
    System.out.println("Size: " + f.length() + " bytes");
    System.out.println("Is directory? " + f.isDirectory());
} else {
    System.out.println("File does not exist yet.");
}
```

Listing Directory Contents

```
File dir = new File(".");     // current directory
String[] contents = dir.list();
for (String name : contents) {
    System.out.println(name);
}

File[] filesOnly = dir.listFiles(File::isFile);  // filter to
    files only
```

Absolute vs Relative File Paths

Relative Path	**Absolute Path**
Interpreted relative to the program's current working directory	Complete path from the filesystem root
e.g. `"student.txt"` or `"data/scores.txt"`	e.g. `"C:/Users/Student/student.txt"`
More portable across machines	Unambiguous but machine-specific

- `getAbsolutePath()` converts any relative path into its full absolute form

Common File-Handling Pitfalls

- Assuming `new File(path)` creates an actual file on disk (only creates object)
- Forgetting that most `File` and stream operations throw `IOException`
- Using the wrong path separator across OSs (prefer `File.separator` or `/`)
- Not checking `exists()` before operating on a file

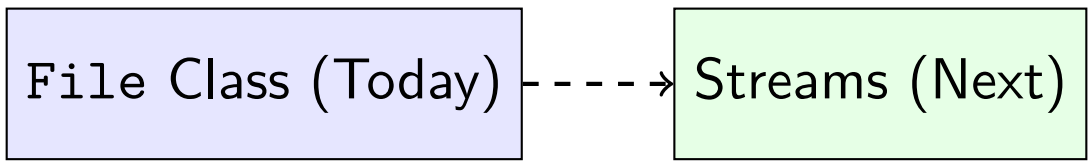
Checklist to Avoid Mistakes:
✓ Call `createNewFile()`
✓ Use try-catch for `IOException`
✓ Use `/` for paths
✓ Verify with `exists()`

Key Takeaways

- File handling provides persistent storage beyond a running program's memory
- `File` class represents a path; use `createNewFile()`, `delete()`, `mkdir()` for operations
- Most `java.io` operations throw checked `IOException`
- Understanding absolute vs relative paths & checking `exists()` prevents bugs

Next Lecture Preview

- Lecture 43 builds on today's File class foundation
- Covers byte streams versus character streams
- Explains reading and writing text file content
- Key classes: `FileReader`, `FileWriter`, and `BufferedReader`



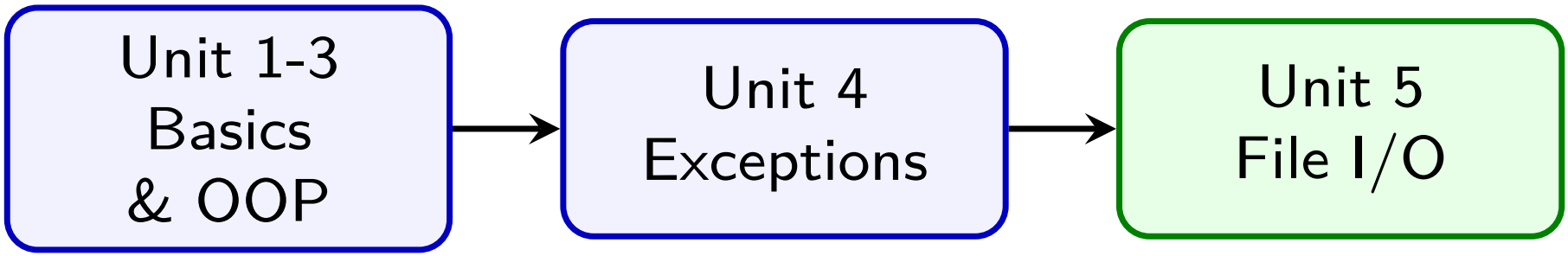
Questions?

Recap: File Handling Fundamentals and the File Class

- Lecture 42 introduced the `File` class for representing, creating, inspecting, and deleting files and directories
- It covered methods like `exists()`, `createNewFile()`, `length()`, `isDirectory()`, and `listFiles()`
- It ended by noting that `File` manages file METADATA and existence, but cannot read or write the actual text content inside a file

Course Overview & Today's Agenda

- Part 1: Byte streams vs character streams - two families of I/O classes
- Part 2: Writing text files with `FileWriter`
- Part 3: Reading text files with `FileReader` and `BufferedReader`



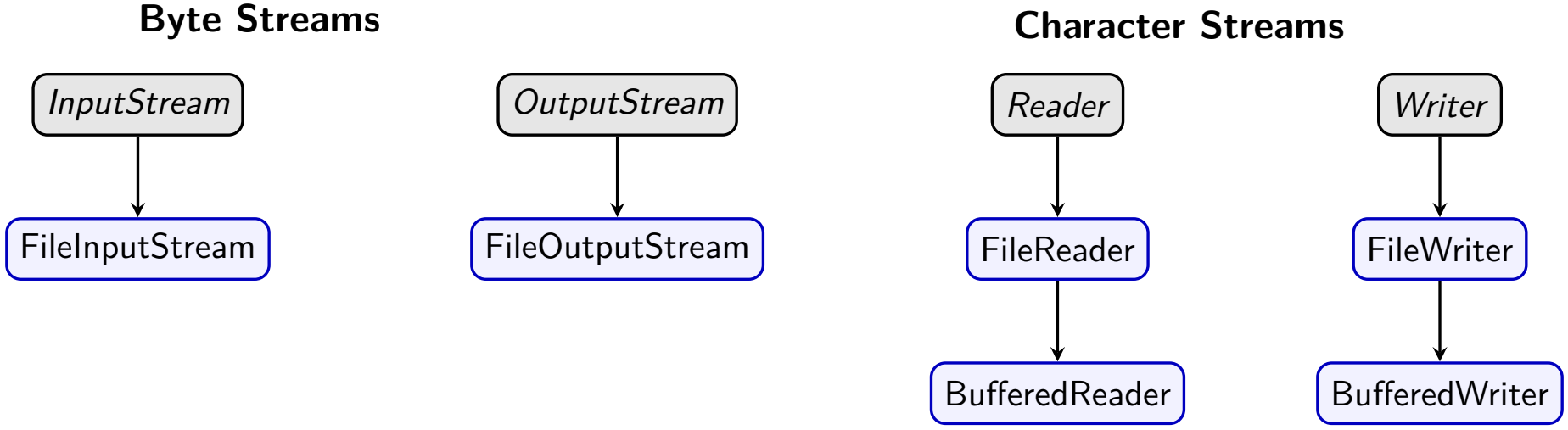
Byte Streams vs Character Streams

- Rule of thumb: use byte streams for binary files, character streams for text files
- Both families are part of `java.io`, introduced in Lecture 42

Byte Streams	**Character Streams**
`InputStream/OutputStream`	`Reader/Writer` family
Read/write raw 8-bit bytes	Read/write 16-bit Unicode characters
Suitable for ANY file type (images, audio)	Suitable specifically for TEXT files
No automatic character encoding	Automatic character encoding handling

The Stream Class Hierarchies

- 'Buffered' classes reduce actual disk accesses by reading/writing in larger chunks internally



Writing Text with `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

try (FileWriter fw = new FileWriter("notes.txt")) {
    fw.write("Java Programming\n");
    fw.write("Unit 5: Collections and File Handling\n");
} catch (IOException e) {
    System.out.println("Write failed: " + e.getMessage());
}
```

Result (`notes.txt`)

Java Programming
Unit 5: Collections and File Handling

try-with-resources: Automatic Stream Cleanup

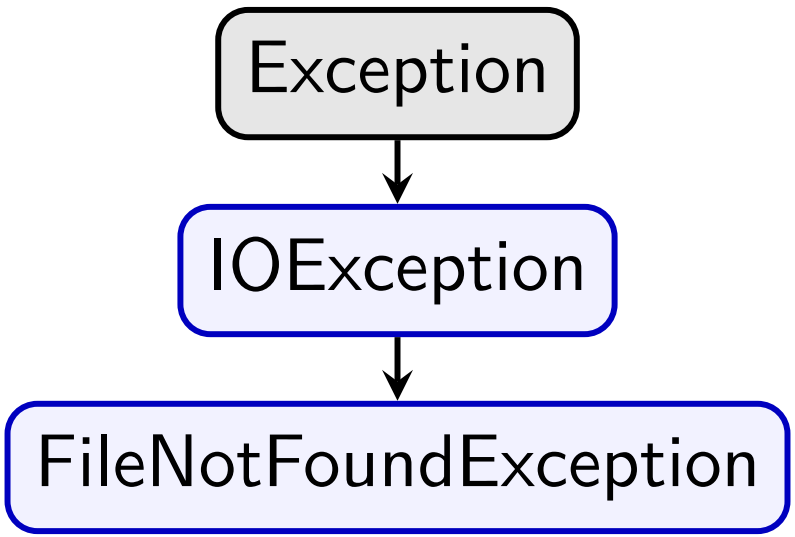
- **try-with-resources** (`try (Resource r = ...) { }`) automatically calls `close()` when the block ends, even if an exception occurs
- Before Java 7, streams had to be closed manually in a `finally` block
- Any class implementing the `AutoCloseable` interface (File streams all do) can be used this way
- Forgetting to close a stream can leak operating system file handles and cause file corruption or data loss

Reading Text with FileReader and BufferedReader

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

try (BufferedReader br = new BufferedReader(new FileReader("
    notes.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("Read failed: " + e.getMessage());
}
```

Exceptions in File I/O

- `IOException`: general checked exception for any input/output failure (disk full, permission denied, device error)
- `FileNotFoundException`: subclass of `IOException`, thrown when a file to be READ does not exist
- Both are **CHECKED exceptions**, forcing a try-catch or a throws declaration
- **Best practice:** catch `FileNotFoundException` before `IOException`



Practice: Copying a File Line by Line

```
try (BufferedReader br = new BufferedReader(new FileReader("
    notes.txt"));
     FileWriter fw = new FileWriter("notes_copy.txt")) {

    String line;
    while ((line = br.readLine()) != null) {
        fw.write(line + "\n");
    }
    System.out.println("File copied successfully.");

} catch (IOException e) {
    System.out.println("Copy failed: " + e.getMessage());
}
```

Appending to an Existing File

```
// The two-argument FileWriter constructor enables append mode
try (FileWriter fw = new FileWriter("log.txt", true)) {
    fw.write("New log entry recorded.\n");
} catch (IOException e) {
    System.out.println("Append failed: " + e.getMessage());
}
// Running this program multiple times ADDS a new line each time
    ,
// instead of erasing old content
```

Common File I/O Pitfalls

- Forgetting that `FileWriter`'s default constructor overwrites existing content (use append constructor)
- Not closing streams (or not using try-with-resources), risking file corruption or resource leaks
- Confusing byte streams and character streams for the wrong file type
- Catching `IOException` too broadly without distinguishing `FileNotFoundException`

Key Takeaways

- Byte streams (`InputStream`/`OutputStream`) handle raw binary data; character streams (`Reader`/`Writer`) handle text
- `FileWriter` writes text, while `FileReader` combined with `BufferedReader`'s `readLine()` reads text efficiently line by line
- try-with-resources automatically closes streams even when exceptions occur
- File I/O throws checked `IOException` (and its subclass `FileNotFoundException`)

Next Lecture Preview

- Lecture 44 moves from saving plain text to saving entire Java objects
- Introducing object serialization concepts and the `Serializable` interface as the foundation for object persistence

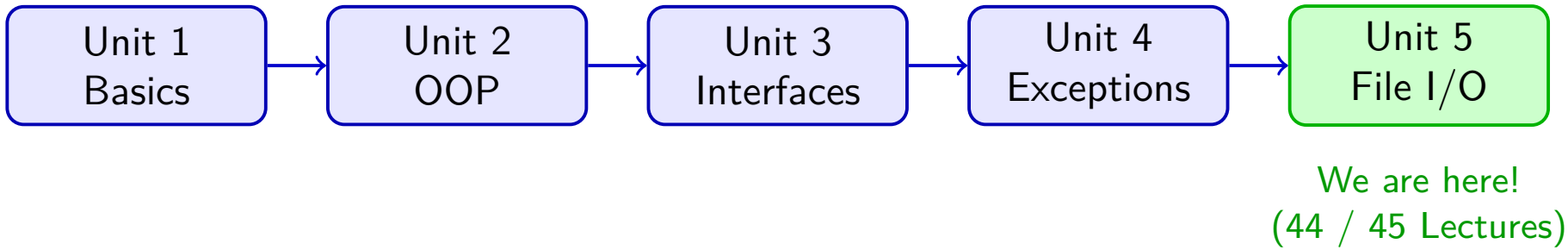
Questions?

Recap: Byte and Character Streams

- Lecture 43 distinguished byte streams (raw binary data) from character streams (text with encoding)
- Demonstrated writing text with `FileWriter` and reading with `FileReader`/`BufferedReader`, in `try-with-resources`
- Connected file I/O's checked exceptions (`IOException`, `FileNotFoundException`) back to exception-handling hierarchy

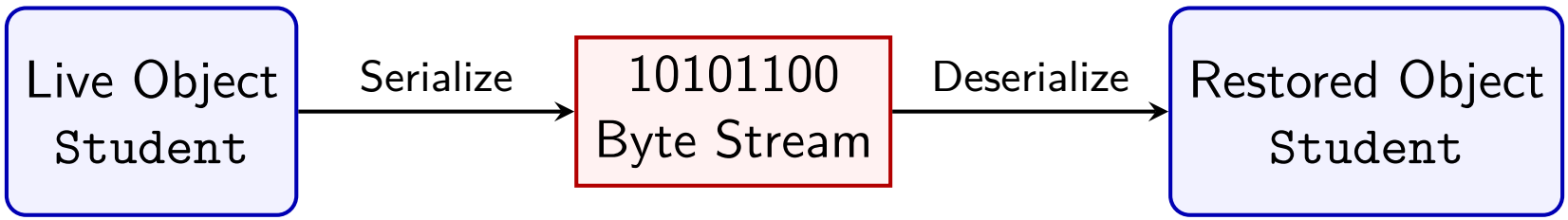
Course Overview & Today's Agenda

- Part 1: What is object serialization, and why do we need it?
- Part 2: The `Serializable` interface - a marker interface
- Part 3: `transient` fields and `serialVersionUID`



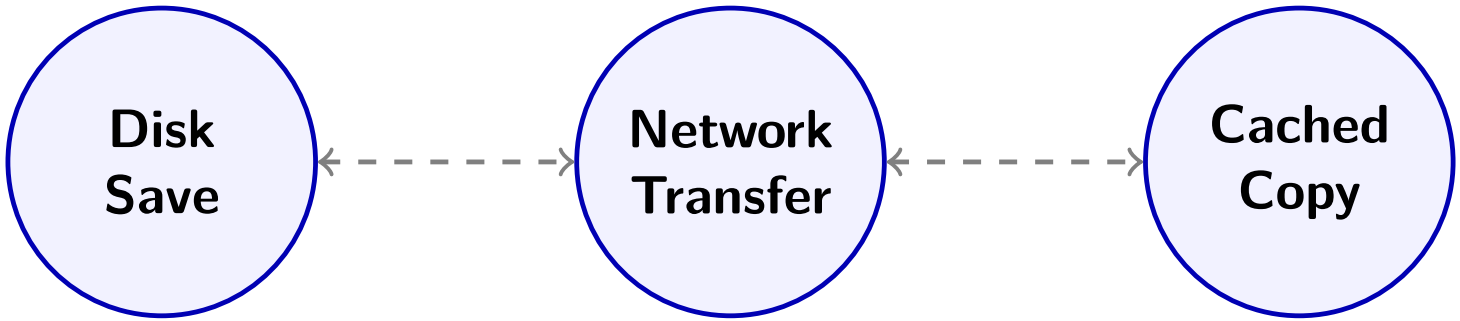
What Is Object Serialization?

- **Serialization**: Converts object's state into a byte stream to save or send.
- **Deserialization**: Reconstructs a live Java object from that byte stream.
- Automates saving objects, preserving exact structure and values.



Why Serialize Objects? Common Use Cases

- **Persistence**: Saving app data (e.g. lists) to survive program restarts.
- **Network transfer**: Sending an object between Java programs over a network.
- **Deep copying**: Duplicating a complex object graph quickly.

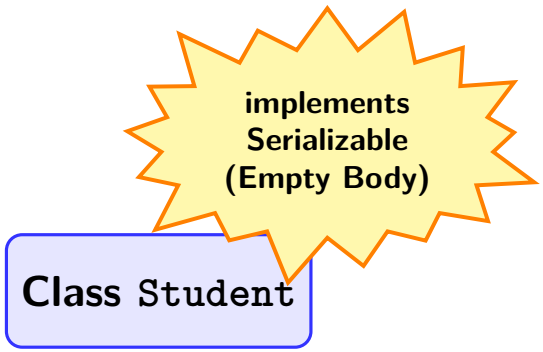


The Serializable Interface: A Marker Interface

Marker Interface

A **MARKER interface** declares NO methods at all. Implementing it simply "tags" the class as eligible for a specific capability.

- To make a class serializable, it must implement `java.io.Serializable`.
- Its purpose is purely to flag capability, not to enforce a method contract.
- Serializing without it throws `NotSerializableException`.



Making a Class Serializable

- The only required change is adding `implements Serializable`.
- No new method implementations are needed!

```
import java.io.Serializable;

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int rollNo;
    double marks;

    Student(String name, int rollNo, double marks) {
        this.name = name;
        this.rollNo = rollNo;
        this.marks = marks;
    }
}
```

serialVersionUID: Why It Matters

- `serialVersionUID` is a unique version identifier used to verify compatibility during deserialization.
- If a class is modified and its ID doesn't match the saved object's ID, it throws `InvalidClassException`.
- Best practice: Always declare explicitly to prevent subtle version-mismatch bugs caused by auto-generation.

The transient Keyword: Excluding Fields

- The `transient` keyword excludes specific fields from serialization.
- Useful for sensitive data (passwords) or temporary cached values.

```
class UserAccount implements Serializable {
    private static final long serialVersionUID = 1L;
    String username;
    transient String password;   // will NOT be serialized

    UserAccount(String u, String p) {
        username = u;
        password = p;
    }
}
// After deserialization: username is restored, but password is
    null
```

What Can and Cannot Be Serialized

- A class is serializable ONLY IF it implements `Serializable` AND every non-`transient`, non-`static` field is also serializable.
- `static` fields belong to the class, not the object, so they are never serialized.
- Composed objects require BOTH classes to be `Serializable`.

Primitives (`int`, `double`)
✓ Serialized

Serializable Objects
✓ Serialized

`transient` fields
✗ Skipped (defaulted)

`static` fields
✗ Skipped

Serialization vs Plain Text File Writing

Plain Text Writing (Lecture 43)	**Serialization (Today)**
Manually convert each field to text	Automatically convert ENTIRE object graph
Manually parse text back on read	Reconstructs object in one call
Human-readable but tedious	Not human-readable but zero manual parsing
Best for logs, CSV, simple data	Best for saving complex Java objects

Designing a Serializable Class: Practice

- Which fields will survive serialization? What will `sessionToken` be?

```
class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int employeeId;
    transient double sessionToken;  // sensitive

    Employee(String name, int id, double token) {
        this.name = name;
        this.employeeId = id;
        this.sessionToken = token;
    }
}
```

Key Takeaways

- **Serialization** converts an object's state into a byte stream for saving/sending; deserialization reverses it.
- `Serializable` is a marker interface with no methods (a distinct use of the interface concept).
- `serialVersionUID` acts as a compatibility fingerprint and should be explicitly declared.
- The `transient` keyword excludes specific fields from serialization (they reset to defaults).

Next Lecture Preview

- Lecture 45 completes serialization with `ObjectOutputStream` and `ObjectInputStream` to perform read/write.
- It is also the final lecture, featuring a full course recap!

Lecture 45:
Streams & Course Recap

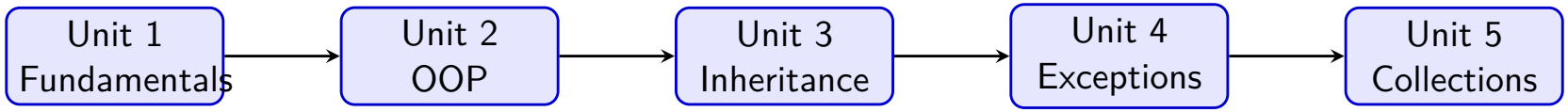
Questions?

Recap: Serialization Concepts and Serializable

- Lecture 44 introduced object serialization - converting an object's state into a byte stream for storage or transfer - and its reverse, deserialization
- It covered the `Serializable` marker interface (no methods, just a tag), `serialVersionUID` for version compatibility, and the `transient` keyword for excluding fields
- Today we complete the picture with the actual classes that perform serialization: `ObjectOutputStream` and `ObjectInputStream`

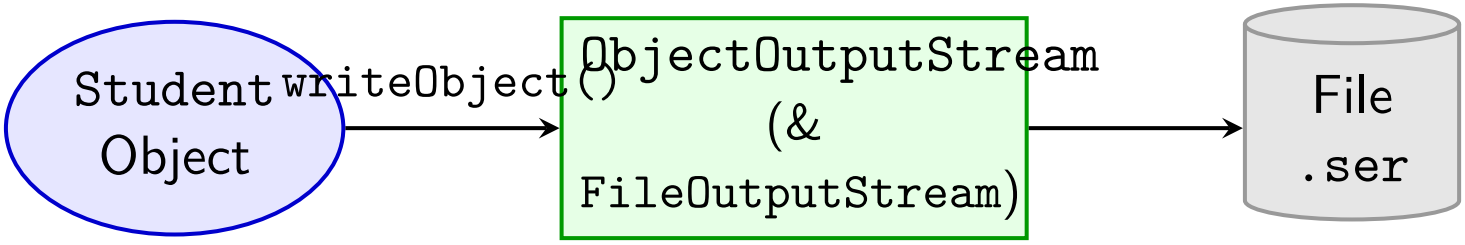
Course Overview & Today's Agenda

- Part 1: `ObjectOutputStream` and `ObjectInputStream` - serializing and deserializing objects
- Part 2: Common serialization pitfalls and their exceptions
- Part 3: Full Course Recap - all 5 units of DI03032021



ObjectOutputStream: Writing an Object to a File

- Wraps a lower-level byte stream (typically `FileOutputStream`) and adds the ability to write whole objects
- Core method: `writeObject(Object obj)` - serializes object and writes bytes to stream
- Requires object's class to implement `Serializable` - else `NotSerializableException`
- Import: `java.io.ObjectOutputStream`, `java.io.FileOutputStream`

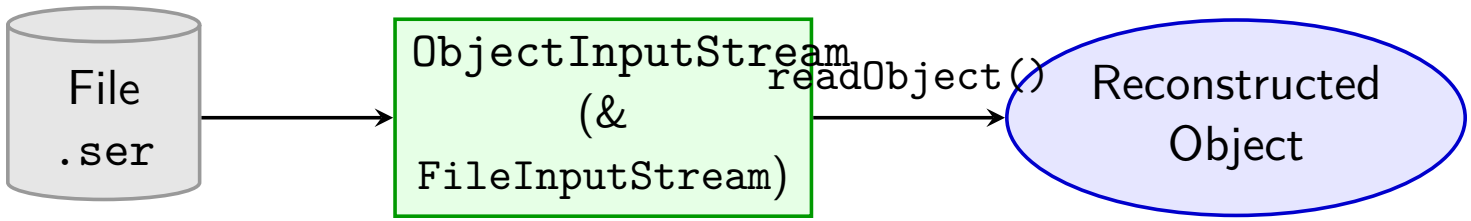


Writing an Object: Code Example

```
import java.io.*;

Student s1 = new Student("Asha", 101, 88.5);
try (ObjectOutputStream oos =
         new ObjectOutputStream(new FileOutputStream("student.
             ser"))) {
    oos.writeObject(s1);
    System.out.println("Student object serialized successfully."
        );
} catch (IOException e) {
    System.out.println("Serialization failed: " + e.getMessage()
        );
}
```

ObjectInputStream: Reading an Object Back

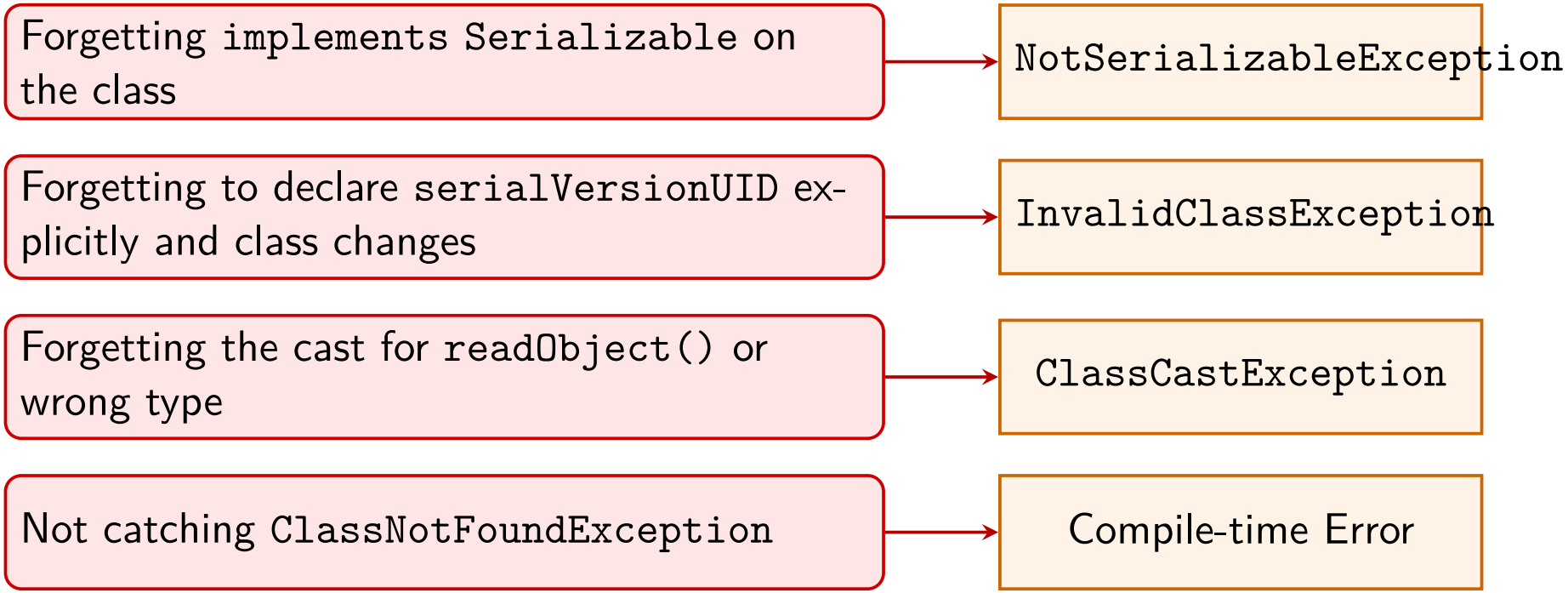
- Wraps lower-level byte stream (typically `FileInputStream`) to read objects back
- Core method: `readObject()` - reads bytes and reconstructs the original object
- Because it returns `Object`, the result must be cast, e.g. `(Student) ois.readObject()`
- Throws TWO checked exceptions: `IOException` (file/stream) & `ClassNotFoundException` (class missing)



Reading an Object Back: Code Example

```
try (ObjectInputStream ois =
        new ObjectInputStream(new FileInputStream("student.ser"
            ))) {
    Student loaded = (Student) ois.readObject();
    System.out.println(loaded.name + ", Roll No: " + loaded.
        rollNo
                        + ", Marks: " + loaded.marks);
} catch (IOException | ClassNotFoundException e) {
    System.out.println("Deserialization failed: " + e.getMessage
        ());
}
```

Common Serialization Pitfalls

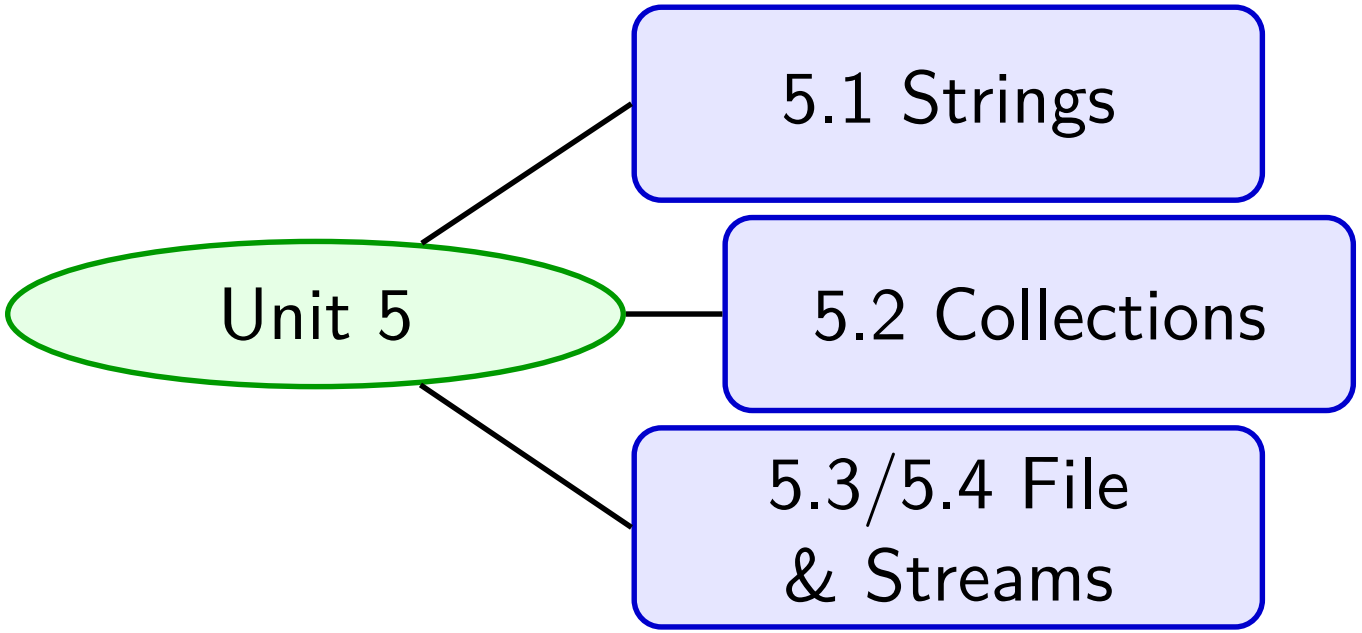


Serializing a Collection of Objects

```
List<Student> students = new ArrayList<>();
students.add(new Student("Asha", 101, 88.5));
students.add(new Student("Ravi", 102, 76.0));

try (ObjectOutputStream oos =
         new ObjectOutputStream(new FileOutputStream("students.
             ser"))) {
    oos.writeObject(students);   // ArrayList itself implements
        Serializable
}
```

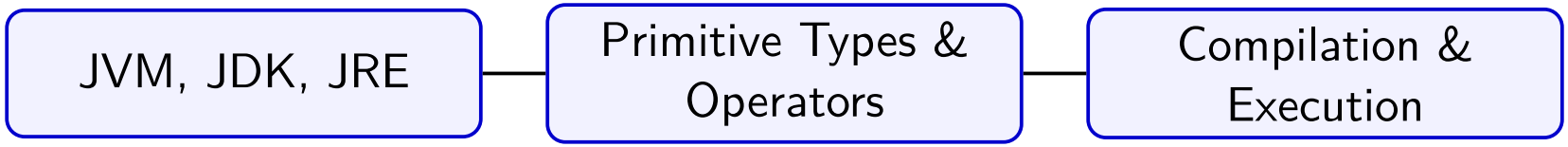
Unit 5 Recap: The Big Picture

- 5.1 String manipulation: `String` class, immutability, `StringBuilder`, regex
- 5.2 Collections Framework: `List`, `Set`, `Map`, and `Iterator`
- 5.3 File handling: `File` class, byte streams vs character streams
- 5.4 Serialization: `Serializable`, `serialVersionUID`, object streams



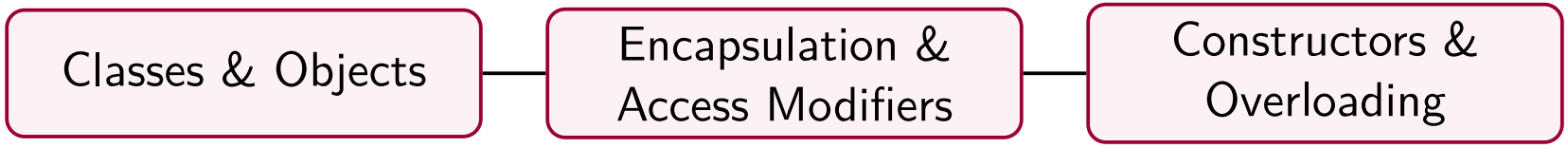
FULL COURSE RECAP: Unit 1

- **Unit 1 - Introduction to Java Programming**
- Covered Java's history, key features, and JVM/JDK/JRE architecture
- Covered environment setup, program structure, compilation (`javac`), and execution (`java`)
- Covered primitive data types, type casting, operators, and operator precedence



FULL COURSE RECAP: Unit 2

- **Unit 2 - Object-Oriented Programming Concepts**
- Covered OOP fundamentals, classes, objects, memory allocation, and GC
- Covered access modifiers, encapsulation, and getter/setter design
- Covered methods, `this` keyword, constructors, overloading, `static` and `final`

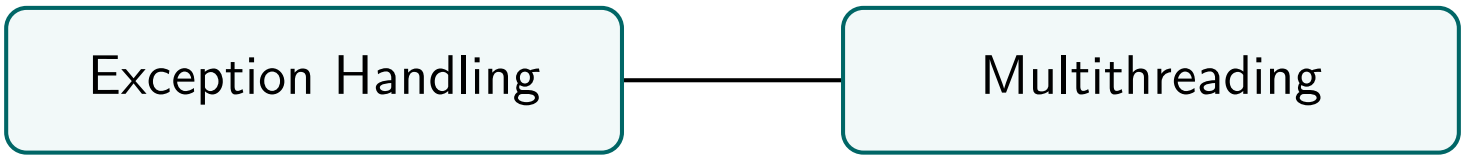


FULL COURSE RECAP: Unit 3

- **Unit 3 - Inheritance, Interfaces, and Packages**
- Covered inheritance (`extends`/`super`), method overriding, dynamic dispatch
- Covered `Object` class methods, interfaces, multiple inheritance via interfaces
- Covered abstract classes versus interfaces, and packages and imports

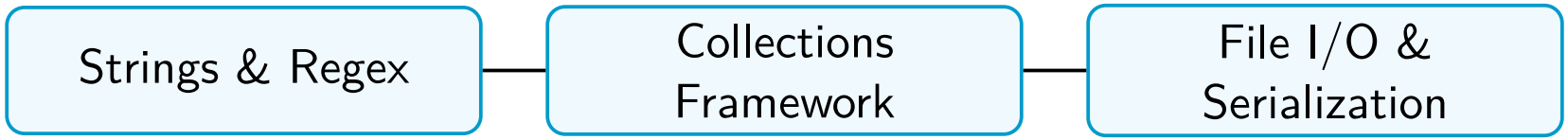
FULL COURSE RECAP: Unit 4

- **Unit 4 - Exception Handling and Multithreading**
- Covered exception hierarchy, `try-catch-finally`, `throw`/`throws`, custom exceptions
- Covered multithreading concepts, process versus thread, and thread lifecycle
- Covered thread creation via `Thread` class and `Runnable` interface



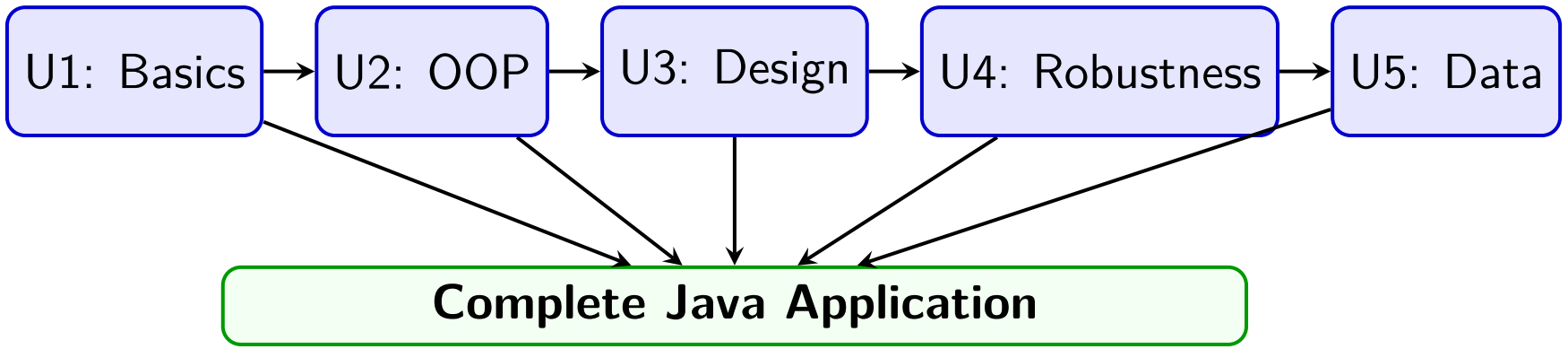
FULL COURSE RECAP: Unit 5

- **Unit 5 - Collections and File Handling**
- Covered `String` class, immutability, `StringBuilder`, regular expressions
- Covered Collections Framework - `List`, `Set`, `Map`, and `Iterator`
- Covered file handling, byte/character streams, object serialization



The Complete Java Programming Picture

- Each unit built directly on the concepts of the previous one
- Together, they form a complete foundation for writing robust and data-capable apps



Course Key Takeaways

- Java's platform independence, strong typing, and OOP model (Unit 1-2) give programs a solid, reusable structure
- Inheritance, interfaces, and packages (Unit 3) let that structure be extended and organised without duplicating code
- Exception handling and multithreading (Unit 4) make programs robust and capable
- The Collections Framework, file handling, and serialization (Unit 5) turn those programs into ones that can manage real, persistent data
- Every concept connects into one coherent picture of professional Java applications

Congratulations and Course Closing

Course Complete

- You have completed all 45 lectures of Java Programming (DI03032021)
- You now understand fundamentals, OOP, inheritance, exceptions, multithreading, Collections, files, and serialization
- This foundation prepares you for further study in web and enterprise Java, Android app development, and Spring Boot
- Thank you for your engagement and hard work!

Questions?