Java Programming – Comprehensive Lecture Notes

From "Learn Java in One Video" Tutorial (Chapters 1–19)

Chapter 1: Your First Java Program

Setup Requirements

To start coding in Java you need two things:

Tool	Purpose
JDK (Java Development Kit)	Contains a **compiler** that converts your source code (`.java`) into **bytecode** (`.class`) that runs on any machine
IDE (Integrated Development Environment)	A workspace to write, edit, and run code (e.g., IntelliJ IDEA Community Edition — free)

Creating a Project (IntelliJ)

1. Open IntelliJ → **New Project**
2. Name it (e.g., `MyFirstProject`)
3. Select the latest JDK version
4. **Uncheck** "Add sample code" (we'll write it ourselves)
5. Click **Create**
6. Navigate to the `src/` folder → **File** → **New** → **Java Class** → name it `Main`

The `main` Method

Every Java program needs a `main` method — it is the **entry point** of the application:

```
public class Main {
    public static void main(String[] args) {
        // Your code goes here
    }
}
```

> *[!IMPORTANT] Without the `main` method, the program **cannot run**. Think of it as a "magic spell" that must be present.*

Printing Output

Method	Behavior
`System.out.print("text")`	Prints text, cursor stays on **same line**
`System.out.println("text")`	Prints text, cursor moves to **next line**

```
System.out.println("I like pizza");
System.out.println("It's really good");
```

```java
System.out.print("Buy me pizza");
```

Escape sequence for a new line: `\n` (works inside any print statement)

Comments

```java
// This is a single-line comment

/*
   This is a
   multi-line comment
*/
```

Comments are **not** displayed as output — they're notes for humans reading the code.

IntelliJ Shortcut

Type `sout` then press **Tab** → auto-generates `System.out.println();`

Chapter 2: Variables

What Is a Variable?

A **variable** is a reusable container for a value. It behaves as if it *is* the value it contains.

Two Categories

Category	Stored	Memory Location	Analogy
Primitive	Direct value	Stack	Handing someone $10
Reference	Memory address pointing to data	Stack → Heap	Giving an IOU that says "$10 at the bank"

Two Steps to Create a Variable

1. **Declaration** — specify the data type and name: `int age;`
2. **Assignment** — give it a value: `age = 21;`
   - Can combine: `int age = 21;`

Primitive Data Types (Beginner Set)

Type	Description	Example
`int`	Whole numbers	`int year = 2025;`
`double`	Numbers with decimals	`double price = 19.99;`
`char`	Single character (single quotes)	`char grade = 'A';`
`boolean`	`true` or `false`	`boolean isStudent = true;`

> *[!NOTE] There are more types (`float` , `long` , etc.) but these four are sufficient for beginners.*

Reference Data Type: `String`

A **String** is a series of characters enclosed in **double quotes**:

```
String name = "Bro Code";
String food = "pizza";
String email = "fake123@gmail.com";
```

String Concatenation

Combine strings and variables using `+` :

```
System.out.println("Hello " + name);                  // Hello Bro Code
System.out.println("Your choice is a " + color + " " + year + " " + car);
```

> *[!TIP] Make sure the variable is **outside** the quotes. `"year"` prints the literal word; `year` (no quotes) prints the variable's value.*

Naming Convention: camelCase

If a variable name has multiple words, capitalize the first letter of each word *after* the first: `isStudent` , `firstName` , `forSale`

Booleans with `if` Statements (Preview)

```
if (isStudent) {
    System.out.println("You are a student");
} else {
    System.out.println("You are not a student");
}
```

Chapter 3: User Input (Scanner)

Importing the Scanner

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

Creating & Closing a Scanner

```
Scanner scanner = new Scanner(System.in);
// ... use scanner ...
scanner.close();  // Always close when done!
```

Reading Different Data Types

Data Type	Scanner Method
`String` (full line)	`scanner.nextLine()`
`String` (single word)	`scanner.next()`
`int`	`scanner.nextInt()`
`double`	`scanner.nextDouble()`
`boolean`	`scanner.nextBoolean()`

Example

```java
System.out.print("Enter your name: ");
String name = scanner.nextLine();

System.out.print("Enter your age: ");
int age = scanner.nextInt();

System.out.println("Hello " + name);
System.out.println("You are " + age + " years old");
```

⚠️ Common Bug: Newline Left in Buffer

When reading an `int` or `double` **then** a `String`, the leftover `\n` character gets consumed by `nextLine()`:

```java
int age = scanner.nextInt();
scanner.nextLine();               // ← Add this to clear the buffer
String color = scanner.nextLine(); // Now works correctly
```

Exercise: Area of a Rectangle

```java
double width = 0, height = 0, area = 0;

System.out.print("Enter the width: ");
width = scanner.nextDouble();

System.out.print("Enter the height: ");
height = scanner.nextDouble();

area = width * height;
System.out.println("The area is: " + area + " cm²");
```

Chapter 4: Mad Libs Game (Project)

A game where the user fills in words (adjective, noun, verb, etc.) and they're inserted into a story.

Key Concepts Practiced

- Declaring multiple `String` variables
- Accepting **user input** with `scanner.nextLine()`
- **String concatenation** to build the story

Code Structure

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        String adjective1, noun1, adjective2, verb1, adjective3;

        System.out.print("Enter an adjective (description): ");
        adjective1 = scanner.nextLine();
        // ... repeat for all variables ...

        System.out.println("\nToday I went to a " + adjective1 + " zoo.");
        System.out.println("In an exhibit, I saw a " + noun1 + ".");
        System.out.println(noun1 + " was " + adjective2 + " and " + verb1 + ".");
        System.out.println("I was " + adjective3 + ".");

        scanner.close();
    }
}
```

Chapter 5: Arithmetic Operators

Basic Operators

Operator	Symbol	Example	Result
Addition	`+`	`10 + 2`	`12`
Subtraction	`-`	`10 - 2`	`8`
Multiplication	`*`	`10 * 2`	`20`
Division	`/`	`10 / 2`	`5`
Modulus (remainder)	`%`	`10 % 3`	`1`

> *[!WARNING] **Integer division** truncates the decimal: `10 / 3` → `3` (not `3.33`). Use `double` or divide by `2.0` to retain decimals.*

Augmented Assignment Operators

Shorthand for modifying and reassigning a variable:

Long form	Shorthand
`x = x + y`	`x += y`
`x = x - y`	`x -= y`
`x = x * y`	`x *= y`
`x = x / y`	`x /= y`
`x = x % y`	`x %= y`

Increment & Decrement

```
x++;  // x = x + 1
x--;  // x = x - 1
```

Commonly used in loops.

Order of Operations — PEMDAS

Parentheses → **E**xponents → **M**ultiplication → **D**ivision → **A**ddition → **S**ubtraction

```
double result = 3 + 4 * (7 - 5) / 2.0;  // = 7.0
```

Evaluation: `(7-5)=2` → `4*2=8` → `8/2.0=4.0` → `3+4.0=7.0`

Chapter 6: Shopping Cart Program (Project)

Concepts Practiced

- Scanner for user input (`nextLine` , `nextDouble` , `nextInt`)
- Arithmetic (total = price × quantity)
- String concatenation for output

Code Structure

```
Scanner scanner = new Scanner(System.in);
String item;
double price, total;
int quantity;
char currency = '$';

System.out.print("What item would you like to buy? ");
item = scanner.nextLine();

System.out.print("What is the price for each? ");
price = scanner.nextDouble();

System.out.print("How many would you like? ");
quantity = scanner.nextInt();
```

```
total = price * quantity;

System.out.println("\nYou have bought " + quantity + " " + item + "/s");
System.out.println("Your total is: " + currency + total);

scanner.close();
```

Chapter 7: If Statements

Syntax

```
if (condition) {
    // code if true
} else if (anotherCondition) {
    // code if the second condition is true
} else {
    // code if none of the above are true (default)
}
```

Comparison Operators

Operator	Meaning
`==`	Equal to (comparison)
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

> *[!CAUTION] `=` is **assignment**, `==` is **comparison**. Using `=` inside an `if` condition is a common beginner mistake.*

Execution Order

Conditions are checked **top-down**. The first `true` condition executes; all subsequent ones are skipped. Ordering matters!

```
// WRONG order:
if (age >= 18) { ... }         // 70-year-old match here and skip below
else if (age >= 65) { ... }    // Never reached for 70!

// CORRECT order:
if (age >= 65) { ... }         // Check senior first
else if (age >= 18) { ... }    // Then adult
```

Comparing Strings

Use `.isEmpty()` to check for an empty string:

```java
if (name.isEmpty()) {
    System.out.println("You didn't enter your name.");
}
```

Using Booleans Directly

```java
boolean isStudent = true;

if (isStudent) {               // No need for: isStudent == true
    System.out.println("You are a student");
}
```

Chapter 8: Random Numbers

Importing & Creating

```java
import java.util.Random;
Random random = new Random();
```

Generating Random Values

Method	Range
`random.nextInt()`	Any int (≈ -2 billion to +2 billion)
`random.nextInt(1, 7)`	`1` to `6` (first inclusive, second **exclusive**)
`random.nextDouble()`	`0.0` to `1.0`
`random.nextBoolean()`	`true` or `false`

Example: Simulating a Coin Flip

```java
boolean isHeads = random.nextBoolean();

if (isHeads) {
    System.out.println("Heads");
} else {
    System.out.println("Tails");
}
```

Chapter 9: Math Class

Constants

Constant	Access	Value
π (pi)	`Math.PI`	3.14159…
e (Euler's number)	`Math.E`	2.71828…

Useful Methods

Method	Description	Example
`Math.pow(base, exp)`	Raise to a power	`Math.pow(2, 3)` → `8.0`
`Math.abs(x)`	Absolute value	`Math.abs(-5)` → `5`
`Math.sqrt(x)`	Square root	`Math.sqrt(9)` → `3.0`
`Math.round(x)`	Round to nearest int	`Math.round(3.14)` → `3`
`Math.ceil(x)`	Round up (ceiling)	`Math.ceil(3.14)` → `4.0`
`Math.floor(x)`	Round down (floor)	`Math.floor(3.99)` → `3.0`
`Math.max(a, b)`	Maximum of two values	`Math.max(10, 20)` → `20`
`Math.min(a, b)`	Minimum of two values	`Math.min(10, 20)` → `10`

Exercise: Hypotenuse of a Right Triangle

Formula: `c = √(a² + b²)`

```
double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
```

Exercise: Circle/Sphere Calculations

```
double circumference = 2 * Math.PI * radius;
double area          = Math.PI * Math.pow(radius, 2);
double volume        = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);
```

Chapter 10: `printf` (Formatted Output)

Format Specifiers

Specifier	Data Type	Example
`%s`	String	`printf("Hello %s", name)`
`%c`	char	`printf("Grade: %c", letter)`
`%d`	int	`printf("Age: %d", age)`

`%f`	double / float	`printf("Price: %f", price)`
`%b`	boolean	`printf("Active: %b", flag)`

> *[!IMPORTANT] Unlike `println` , `printf` does **not** add a newline automatically. Add `\n` at the end: `printf("Hello %s\n", name);`*

Precision (Limiting Decimal Places)

```
System.out.printf("Price: %.2f\n", 9.99);    // Price: 9.99
System.out.printf("PI: %.1f\n", Math.PI);     // PI: 3.1
```

Flags

Flag	Effect	Example
`+`	Show + for positive numbers	`%+.2f` → `+9.99`
`,`	Thousands separator	`%,.2f` → `9,999.00`
`(`	Negative in parentheses	`%(f` → `(54.10)`
(space)	Space before positive numbers	`% f` → `9.99`

Width & Padding

Syntax	Effect
`%04d`	Zero-pad to 4 digits: `0001`
`%4d`	Right-justify in 4-char field
`%-4d`	Left-justify in 4-char field

Chapter 11: Compound Interest Calculator (Project)

Formula

`A = P × (1 + r/n)^(n×t)`

Variable	Meaning
`P`	Principal (initial investment)
`r`	Annual interest rate (decimal)
`n`	Times compounded per year
`t`	Number of years
`A`	Final amount

Implementation

```
double principal, rate, amount;
int timesCompounded, years;

// Accept user input for all variables...

rate = rate / 100;  // Convert percentage to decimal

amount = principal * Math.pow(1 + rate / timesCompounded,
                              timesCompounded * years);

System.out.printf("The amount after %d years is $%.2f\n", years, amount);
```

Chapter 12: Nested If Statements 🎟️

Concept

If statements **inside** other if statements — check a second condition only after the first is true.

Example: Movie Ticket Discounts

Condition	Discount
Student only	10%
Senior only	20%
Student **and** Senior	30%
Neither	Full price

```
boolean isStudent = true;
boolean isSenior = false;
double price = 9.99;

if (isStudent) {
    if (isSenior) {
        // Both discounts
        System.out.println("Senior discount 20% + Student discount 10%");
        price *= 0.7;
    } else {
        // Student only
        System.out.println("Student discount 10%");
        price *= 0.9;
    }
} else {
    if (isSenior) {
        // Senior only
        System.out.println("Senior discount 20%");
        price *= 0.8;
    }
}

System.out.printf("The price of a ticket is $%.2f\n", price);
```

Chapter 13: String Methods

Quick Reference

Method	Returns	Description
`.length()`	`int`	Number of characters
`.charAt(index)`	`char`	Character at given index (0-based)
`.indexOf(str)`	`int`	First index of a character/substring
`.lastIndexOf(str)`	`int`	Last index of a character/substring
`.toUpperCase()`	`String`	All characters uppercase
`.toLowerCase()`	`String`	All characters lowercase
`.trim()`	`String`	Removes leading/trailing whitespace
`.replace(old, new)`	`String`	Replace characters
`.isEmpty()`	`boolean`	True if string has length 0
`.contains(str)`	`boolean`	True if string contains the given text
`.equals(str)`	`boolean`	True if strings have identical characters
`.equalsIgnoreCase(str)`	`boolean`	Same as equals, case-insensitive

Examples

```
String name = "Bro Code";

name.length();               // 8
name.charAt(0);              // 'B'
name.indexOf("o");           // 2  (first 'o')
name.lastIndexOf("o");       // 5  (last 'o')
name.toUpperCase();          // "BRO CODE"
name.toLowerCase();          // "bro code"
name.replace('o', 'a');      // "Bra Cade"
name.isEmpty();              // false
name.contains(" ");          // true
name.equals("password");     // false
```

Chapter 14: Substrings

The `substring()` Method

Extracts a portion of a string.

Signature	Behavior
`substring(start)`	From `start` index to end of string
`substring(start, end)`	From `start` (inclusive) to `end` (**exclusive**)

Email Slicer Program

```java
String email = "bro123@gmail.com";

int atIndex = email.indexOf("@");

String username = email.substring(0, atIndex);       // "bro123"
String domain   = email.substring(atIndex + 1);      // "gmail.com"
```

Validating with `contains()`

```java
if (email.contains("@")) {
    // extract username and domain
} else {
    System.out.println("Emails must contain @");
}
```

Chapter 15: Weight Converter (Project) 🏋️

Key Concepts

- Using `if` / `else if` / `else` for option selection
- Conversion formulas:
  - **Pounds → Kilograms**: `weight × 0.453592`
  - **Kilograms → Pounds**: `weight × 2.2462`

Pseudocode Approach

Using comments as pseudocode before coding is a great habit:

```
// 1. Declare variables
// 2. Welcome message + menu
// 3. Prompt for user choice
// 4. Option 1: lbs → kg
// 5. Option 2: kg → lbs
// 6. Else: invalid choice
```

Implementation

```
if (choice == 1) {
    System.out.print("Enter the weight in lbs: ");
    weight = scanner.nextDouble();
    newWeight = weight * 0.453592;
    System.out.printf("The new weight in kgs is %.1f\n", newWeight);
} else if (choice == 2) {
    System.out.print("Enter the weight in kg: ");
    weight = scanner.nextDouble();
    newWeight = weight * 2.2462;
    System.out.printf("The new weight in lbs is %.1f\n", newWeight);
} else {
    System.out.println("That was not a valid choice");
}
```

Chapter 16: Ternary Operator

Formula

```
variable = (condition) ? valueIfTrue : valueIfFalse;
```

It's a **simpler alternative** to an `if-else` statement.

Examples

Pass or Fail:

```
int score = 75;
String result = (score >= 60) ? "Pass" : "Fail";
```

Even or Odd:

```
int number = 3;
String evenOrOdd = (number % 2 == 0) ? "Even" : "Odd";
```

AM / PM:

```
int hours = 13;
String timeOfDay = (hours < 12) ? "AM" : "PM";
```

Tax Bracket:

```
double income = 60000;
double taxRate = (income >= 40000) ? 0.25 : 0.15;
```

Chapter 17: Temperature Converter (Project) 🌡️

Formulas

- **Fahrenheit → Celsius**: `(temp - 32) × 5/9`
- **Celsius → Fahrenheit**: `(temp × 5/9) + 32`

Using the Ternary Operator

```java
System.out.print("Convert to Celsius or Fahrenheit? (C/F): ");
String unit = scanner.next().toUpperCase();   // Method chaining!

double newTemp = (unit.equals("C"))
    ? (temp - 32) * 5.0 / 9
    : (temp * 9.0 / 5) + 32;

System.out.printf("%.1f°%s\n", newTemp, unit);
```

> *[!TIP] **Method chaining**: `scanner.next().toUpperCase()` calls two methods in sequence — gets the input, then converts it to uppercase in one line.*

Chapter 18: Enhanced Switches

Why Use Switches?

When you have **many `else if` statements** checking the same variable against different values, a `switch` is cleaner and more efficient.

Syntax (Enhanced — Java 14+)

```java
switch (variable) {
    case value1 -> {
        // code
    }
    case value2, value3 -> {
        // shared code for multiple cases
    }
    default -> {
        // if no cases match
    }
}
```

Example: Day of the Week

```java
switch (day) {
    case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"
        -> System.out.println("It is a weekday");
    case "Saturday", "Sunday"
        -> System.out.println("It is the weekend");
    default
```

```
        -> System.out.println(day + " is not a day");
}
```

Key Features of Enhanced Switches

- Uses **arrow operator** (`->`) instead of colon
- **No fall-through** — no `break` statements needed
- Multiple cases can be **comma-separated**
- `default` acts like `else`

Chapter 19: Calculator Program (Project)

Concepts Practiced

- Scanner input (`nextDouble` , `nextChar`)
- Enhanced `switch` for operator selection
- `Math.pow()` for exponentiation

Code Structure

```
Scanner scanner = new Scanner(System.in);
double num1, num2, result;
char operator;

System.out.print("Enter first number: ");
num1 = scanner.nextDouble();

System.out.print("Enter operator (+, -, *, /, ^): ");
operator = scanner.next().charAt(0);

System.out.print("Enter second number: ");
num2 = scanner.nextDouble();

switch (operator) {
    case '+' -> result = num1 + num2;
    case '-' -> result = num1 - num2;
    case '*' -> result = num1 * num2;
    case '/' -> {
        if (num2 != 0) {
            result = num1 / num2;
        } else {
            System.out.println("Cannot divide by zero!");
            result = 0;
        }
    }
    case '^' -> result = Math.pow(num1, num2);
    default  -> {
        System.out.println("Not a valid operator");
        result = 0;
    }
}
```

```
System.out.println("Result: " + result);
scanner.close();
```

Quick Reference: Key Java Concepts (Chapters 1–19)

Data Types at a Glance

Type	Category	Size	Example
`int`	Primitive	4 bytes	`42`
`double`	Primitive	8 bytes	`3.14`
`char`	Primitive	2 bytes	`'A'`
`boolean`	Primitive	1 bit	`true`
`String`	Reference	varies	`"Hello"`

Common Imports

```
import java.util.Scanner;   // User input
import java.util.Random;    // Random numbers
// Math class — no import needed (java.lang)
```

Common Patterns

```
// Scanner pattern
Scanner scanner = new Scanner(System.in);
// ... use scanner ...
scanner.close();

// Random number in range [min, max]
Random random = new Random();
int num = random.nextInt(min, max + 1);  // max+1 because exclusive

// Formatted output
System.out.printf("%.2f\n", value);      // 2 decimal places
```

Chapter 20: Logical Operators

Timestamp: 3:10:02 **Key Concepts:**

- **AND (`&&`)**: Returns true if *both* conditions are true.
- **OR (`||`)**: Returns true if *at least one* condition is true.
- **NOT (`!`)**: Reverses the boolean value (true becomes false, false becomes true).

Code Example:

```
int temp = 25;
boolean isSunny = true;

// AND Operator
if (temp >= 0 && temp <= 30 && isSunny) {
    System.out.println("The weather is good.");
    System.out.println("It is sunny outside.");
}

// OR Operator
if (temp > 30 || temp < 0) {
    System.out.println("The weather is bad.");
}

// NOT Operator
if (!isSunny) {
    System.out.println("It is cloudy.");
}
```

Project: Username Validator Validates that a username is 4-12 characters long and contains no spaces or underscores.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter username: ");
String username = scanner.nextLine();

if (username.length() < 4 || username.length() > 12) {
    System.out.println("Username must be between 4-12 characters");
} else if (username.contains(" ") || username.contains("_")) {
    System.out.println("Username must not contain spaces or underscores");
} else {
    System.out.println("Welcome " + username);
}
```

Chapter 21: While Loops

Timestamp: 3:21:26 **Key Concepts:**

- `while` **loop**: Repeats code *forever* as long as the condition is true. Checks condition *before* execution.
- `do-while` **loop**: Executes the code block *once*, then checks the condition. Guarantees at least one execution.
- **Infinite Loop**: A loop where the condition never becomes false (e.g., `while(true)`).

Code Example (While Loop): Forcing user to enter input.

```
Scanner scanner = new Scanner(System.in);
String name = "";

while (name.isBlank()) {
    System.out.print("Enter your name: ");
```

```java
    name = scanner.nextLine();
}
System.out.println("Hello " + name);
```

Code Example (Do-While Loop):

```java
Scanner scanner = new Scanner(System.in);
int number;

do {
    System.out.print("Enter a number between 1 and 10: ");
    number = scanner.nextInt();
} while (number < 1 || number > 10);

System.out.println("You picked " + number);
```

Chapter 22: Number Guessing Game (Project)

Timestamp: 3:33:47 **Description:** A game where the user guesses a random number within a range. **Key Features:**

- Uses `Random` class for number generation.
- Uses `do-while` loop to keep game running until correct guess.
- Tells user if guess is "Too high" or "Too low".

Code:

```java
import java.util.Random;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Random random = new Random();
        Scanner scanner = new Scanner(System.in);

        int min = 1;
        int max = 100;
        int randomNumber = random.nextInt(max - min + 1) + min;
        int guess;
        int attempts = 0;

        System.out.println("Number Guessing Game");
        System.out.printf("Guess a number between %d-%d\n", min, max);

        do {
            System.out.print("Enter a guess: ");
            guess = scanner.nextInt();
            attempts++;

            if (guess < randomNumber) {
                System.out.println("Too low!");
            } else if (guess > randomNumber) {
                System.out.println("Too high!");
            } else {
                System.out.println("CORRECT! The number was " + randomNumber);
                System.out.println("# of attempts: " + attempts);
            }
        } while (guess != randomNumber);

        scanner.close();
    }
}
```

Chapter 23: For Loops

Timestamp: 3:43:39 **Key Concepts:**

- `for` **loop**: Executes code a *specific* amount of times.
- Structure: `for (initialization; condition; update) { ... }`
- **Index (** `i` **)**: Commonly used counter variable.

Code Examples:

```
// Count up
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}

// Count down
for (int i = 10; i >= 0; i--) {
    System.out.println(i);
}

// Iterate custom amount from user input
System.out.print("How many times to loop? ");
int max = scanner.nextInt();
for (int i = 0; i < max; i++) {
    System.out.println(i);
}
```

Project: Countdown Timer Uses `Thread.sleep()` to pause execution.

```
System.out.print("How many seconds to countdown? ");
int start = scanner.nextInt();

for (int i = start; i > 0; i--) {
    System.out.println(i);
    Thread.sleep(1000); // Sleep for 1000ms (1 second)
}
System.out.println("Happy New Year!");
```

Chapter 24: Break & Continue

Timestamp: 3:53:35 **Key Concepts:**

- `break` : Exits the loop entirely immediately (Stop).
- `continue` : Skips the *current* iteration and moves to the next one (Skip).

Code Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Stops loop at 5 (prints 0-4)
        // continue; // Skips 5 (prints 0-4, 6-9)
    }
    System.out.println(i);
}
```

Chapter 25: Nested Loops

Timestamp: 3:55:45 **Key Concepts:**

- A loop inside another loop.
- Commonly used for matrices, grids, or 2D patterns.
- Outer loop controls rows, inner loop controls columns.
- Naming convention: outer index `i`, inner index `j`.

Project: Symbol Grid User defines rows, columns, and symbol.

```
System.out.print("Enter rows: ");
int rows = scanner.nextInt();
System.out.print("Enter columns: ");
int columns = scanner.nextInt();
System.out.print("Enter symbol: ");
String symbol = scanner.next();

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
        System.out.print(symbol);
    }
    System.out.println(); // New line after each row
}
```

Chapter 26: Methods

Timestamp: 4:04:29 **Key Concepts:**

- **Method**: A block of reusable code executed when called.
- Follows DRY principle (Don't Repeat Yourself).
- **Parameters**: Variables defined in method declaration to receive values.
- **Arguments**: Actual values passed to method when calling.

- **Return Type**: Data type of value returned (`void` if nothing returned).
- **static**: Required to call from `main` (which is static).

Code Example:

```
public static void main(String[] args) {
    singHappyBirthday("SpongeBob", 30);

    double result = square(3.0);
    System.out.println(result); // 9.0

    System.out.println(checkAge(21)); // true
}

// Method with parameters
static void singHappyBirthday(String name, int age) {
    System.out.println("Happy birthday " + name);
    System.out.printf("You are %d years old\n", age);
}

// Method with return value
static double square(double number) {
    return number * number;
}

// Age check method
static boolean checkAge(int age) {
    if (age >= 18) {
        return true;
    } else {
        return false;
    }
}
```

Chapter 27: Overloaded Methods

Timestamp: 4:19:51 **Key Concepts:**

- Methods sharing the **same name** but different **parameters** (number, type, or order).
- **Method Signature**: Name + Parameters. Must be unique.

Code Example:

```
// Add two numbers
static int add(int a, int b) {
    return a + b;
}

// Add three numbers (Overloaded)
static int add(int a, int b, int c) {
    return a + b + c;
}

// Different types (Overloaded)
static double add(double a, double b) {
    return a + b;
}
```

Note: Return type alone does not distinguish overloaded methods.

Chapter 28: Variable Scope

Timestamp: 4:26:06 **Key Concepts:**

- **Local Scope**: Variables declared *inside* a method/block. Only visible within that block.
- **Class Scope (Global)**: Variables declared *outside* methods (in class). Visible to all methods in class.
- Why? Methods can't see each other's local variables (like neighbors in different houses).

Example:

```
public class Main {
    static int x = 3; // Class scope (visible everywhere)

    public static void main(String[] args) {
        int x = 1; // Local scope (shadows class x)
        System.out.println(x); // Prints 1
        doSomething();
    }

    static void doSomething() {
        int x = 2; // Local scope
        System.out.println(x); // Prints 2 (own x)
        // If local x removed, would print 3 (class x)
    }
}
```

Chapter 29: Banking Program (Project)

Timestamp: 4:30:57 **Description:** A console banking app to Show Balance, Deposit, Withdraw, and Exit. **Key Features:**

- Uses methods to organize code (`showBalance()` , `deposit()` , `withdraw()`).
- Uses `static Scanner` at class level to share across methods.
- Validates inputs (no negative deposits/withdrawals, no overdrafts).

Code Structure:

```
static double balance = 0;
static Scanner scanner = new Scanner(System.in);

public static void main(String[] args) {
    int choice;
    boolean isRunning = true;

    while (isRunning) {
        System.out.println("1. Show Balance\n2. Deposit\n3. Withdraw\n4. Exit");
        System.out.print("Enter choice: ");
        choice = scanner.nextInt();

        switch(choice) {
            case 1 -> showBalance();
            case 2 -> balance += deposit();
            case 3 -> balance -= withdraw();
            case 4 -> isRunning = false;
            default -> System.out.println("Invalid Choice");
        }
    }
}

static void showBalance() {
    System.out.printf("$%.2f\n", balance);
}

static double deposit() {
    System.out.print("Enter amount: ");
    double amount = scanner.nextDouble();
    if (amount < 0) {
        System.out.println("Cannot deposit negative amount");
        return 0;
    }
    return amount;
}
// withdraw() looks similar with overdraft check
```

Chapter 30: Dice Roller Program (Project)

Timestamp: 4:51:27 **Description:** User enters number of dice to roll, program displays total and ASCII art faces.
Key Features:

- `Random` numbers (1-6).
- **ASCII Art**: Stored in strings to visualize dice faces.
- `printDie(int roll)` method prints art based on roll value.

Code Snippet (ASCII Art Logic):

```
static void printDie(int roll) {
    String die1 = """
         -------
        |       |
        |   *   |
        |       |
         -------""";
    // ... define die2, die3, etc.
    switch (roll) {
        case 1 -> System.out.print(die1);
        // ... cases for other dice
    }
}
```

Chapter 31: Arrays

Timestamp: 5:03:33 **Key Concepts:**

- **Array**: Collection of values of the *same data type*.
- Fixed size after creation.
- **Index**: Zero-based (0 to length-1).

Syntax:

```
// Declaration & Initialization
String[] cars = {"Camaro", "Corvette", "Tesla"};

// Access Element
System.out.println(cars[0]); // Camaro

// Modify Element
cars[0] = "Mustang";

// Length Property
System.out.println(cars.length); // 3

// Loop through array
for (int i = 0; i < cars.length; i++) {
    System.out.println(cars[i]);
}
```

Chapter 32: User Input into Array

Timestamp: 5:12:36 **Key Concepts:**

- Must compile/decide size *before* assigning values if not using initializer list.
- Syntax: `String[] foods = new String[size];`

Code Example:

```
System.out.print("How many foods? ");
int size = scanner.nextInt();
scanner.nextLine(); // Clear buffer

String[] foods = new String[size]; // Create array of specific size

for (int i = 0; i < foods.length; i++) {
    System.out.print("Enter food: ");
    foods[i] = scanner.nextLine();
}

// Display
for (String food : foods) { // Enhanced for-loop
    System.out.println(food);
}
```

Chapter 33: Search an Array

Timestamp: 5:20:42 **Key Concepts:**

- **Linear Search**: Iterating through array element by element to find a match.
- For Strings, use `.equals()`, NOT `==`.

Code Example:

```
int[] numbers = {1, 9, 3, 5, 2};
int target = 5;
boolean isFound = false;

for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == target) {
        System.out.println("Found at index: " + i);
        isFound = true;
        break;
    }
}

if (!isFound) {
    System.out.println("Not found");
}
```

Chapter 34: Varargs (Variable Arguments)

Timestamp: 5:28:08 **Key Concepts:**

- Allows a method to accept a varying number of arguments.
- Syntax: `Type... name` (e.g., `int... numbers`).
- Treated as an **array** inside the method.
- **Note**: Vararg parameter must be the specific last parameter in the list.

Code Example:

```
static int add(int... numbers) {
    int sum = 0;
    for (int number : numbers) {
        sum += number;
    }
    return sum;
}
```

```
// Usage
System.out.println(add(1, 2, 3, 4)); // 10
System.out.println(add(5, 10));      // 15
```

Chapter 35: 2D Arrays

Timestamp: 5:34:38 **Key Concepts:**

- An array of arrays (Matrix/Grid).
- Rows and Columns.
- Syntax: `Type[][] name = new Type[rows][cols];` OR `{{}, {}}` .
- Access: `data[row][col]` .

Code Example:

```
// Phone keypad layout
char[][] telephone = {
    {'1', '2', '3'},
    {'4', '5', '6'},
    {'7', '8', '9'},
    {'*', '0', '#'}
};

// Nested loop to display
for (char[] row : telephone) {
    for (char digit : row) {
        System.out.print(digit + " ");
    }
    System.out.println();
}
```

Chapter 36: Project - Quiz Game

Timestamp: ~5:50:00

A console-based multiple-choice quiz game.

Key Concepts:

- **Arrays for Data:** Storing questions and options in arrays.
- **Logic:** Looping through questions, validating input, and tracking score.

Implementation Steps:

1. **Define Questions & Options:**
   - `String[] questions` : Array of question strings.
   - `String[][] options` : 2D array of choices (rows = questions, cols = options).
   - `int[] answers` : Array of indices for correct answers.
2. **Variables:** `score` , `guess` .
3. **Game Loop:**

    - Iterate through questions.
    - Print question.
    - Iterate through options for that question and print them.
    - Get user input (`scanner.nextInt()`).
    - **Validation:** Check if `guess` matches `answers[i]` .
    - **Feedback:** Print "Correct" or "Wrong".
    - Update `score` .

4. **Final Output:** Display `score` out of total questions.

Chapter 37: Project - Rock Paper Scissors

Timestamp: 5:59:13

Classic game against the computer.

Key Concepts:

- **Randomization:** Getting a random choice for the computer.
- **Win Conditions:** Complex `if/else` logic to determine the winner.

Implementation Details:

- **Choices:** Array `{"Rock", "Paper", "Scissors"}` .
- **Computer Choice:** `random.nextInt(3)` used as an index for the array.
- **Input Validation:** `while` loop checks if input is valid (Rock/Paper/Scissors).
- **Win Logic:**
    - **Tie:** Player choice equals Computer choice.
    - **Win:** (Rock vs Scissors) OR (Paper vs Rock) OR (Scissors vs Paper).
    - **Lose:** `else` condition.
- **Play Again:** Encapsulate game logic in a `do-while` loop asking user to continue.

Chapter 38: Project - Slot Machine

Timestamp: 6:15:04

A betting game using emojis as symbols.

Key Concepts:

- **Validation:** Checking balance and validity of bet amount.
- **Methods:** separating logic into `spinRow()` , `printRow()` , `getPayout()` .
- **Switch Expression:** Calculating payout based on symbol type.

Game Logic:

1. **Balance:** Start with `$100` .
2. **Betting:** User enters amount. Verify `bet > 0` and `bet <= balance` .
3. **Spinning:** Generate 3 random symbols (e.g., 🍒 , 🍉 , 🔔) into an array.
4. **Payout:**
    - Check if all 3 symbols match (High payout).
    - Check if first 2 match (Medium payout).
    - **Switch Statement:** Assign multipliers based on the symbol (e.g., Star = 20x, Cherry = 3x).
5. **Update Balance:** Add winnings or subtract bet.
6. **Loop:** Continue until balance is 0 or user quits.

Chapter 39: Object-Oriented Programming (OOP)

Timestamp: 6:41:52

Concept:

- **Object:** A representation of a real-world entity that contains **data** (attributes) and **actions** (methods).
- **Class:** A blueprint or template for creating objects.

Example: Car Class

```
public class Car {
    String make = "Ford";
    String model = "Mustang";
    int year = 2025;
    boolean isRunning = false;

    void start() {
        isRunning = true;
        System.out.println("Engine started");
    }

    void stop() {
        isRunning = false;
        System.out.println("Engine stopped");
    }
}
```

Usage:

- **Instantiation:** `Car myCar = new Car();`
- **Accessing Attributes:** `myCar.model` (Dot operator)
- **Calling Methods:** `myCar.start();`

Chapter 40: Constructors

Timestamp: 6:51:42

A special method inherited via the class name used to **initialize objects** with specific attributes.

Key Points:

- Called automatically when `new` keyword is used.
- Allows creating unique objects (e.g., `Car` with different colors).
- **`this` keyword:** Refers to the *current object*. Used to distinguish between class attributes and constructor parameters (e.g., `this.name = name`).

Example:

```
Human human1 = new Human("Rick", 65, 70.0);
Human human2 = new Human("Morty", 16, 50.0);
```

Chapter 41: Overloaded Constructors

Timestamp: 7:01:50

Multiple constructors in a class with **different parameter lists**.

Usage:

- Allows creating objects in different ways (e.g., a `Pizza` with just "bread" vs. a `Pizza` with "bread", "sauce", "cheese").
- Java differentiates them by the **number** and **type** of arguments passed.

Chapter 42: Array of Objects

Timestamp: 7:08:27

Storing objects within an array, just like primitives.

Syntax:

```
// 1. Declare and Allocate
Food[] refrigerator = new Food[3];

// 2. Initialize
Food food1 = new Food("Pizza");
Food food2 = new Food("Hamburger");
Food food3 = new Food("Hotdog");

// 3. Assign
refrigerator[0] = food1;
refrigerator[1] = food2;
refrigerator[2] = food3;

// Alternatively
Food[] pantry = {new Food("Pizza"), new Food("Burger")};
```

Chapter 43: Static Keyword

Timestamp: 7:14:07

Modifier that makes a variable or method belong to the **class** rather than a specific **instance/object**.

Key Use Cases:

- **Static Variables:** Shared among all instances (e.g., `numberOfFriends` counter that increments every time a `Friend` object is created). If changed, it changes for *all* objects.
- **Static Methods:** Utility methods that don't need access to object data (e.g., `Math.round()`). Called via Class Name (`Friend.displayFriends()`).

Chapter 44: Inheritance

Timestamp: 7:22:06

Mechanism where a child class acquires the attributes and methods of a parent class.

Key Points:

- **Keyword:** `extends`
- **Parent (Super) Class:** The general class (e.g., `Vehicle`).
- **Child (Sub) Class:** The specific class (e.g., `Car` , `Bicycle`).
- **Benefits:** Code reusability. Update logic in one place (Parent) to affect all children.

Example:

```
public class Car extends Vehicle {
    // Car inherits everything from Vehicle
    // Can also add its own specific fields/methods
}
```

Chapter 45: Super Keyword

Timestamp: 7:31:12

Refers to the **parent class** (superclass) of an object.

Use Cases:

- **Constructors:** `super(arguments)` calls the parent constructor. Essential when the parent constructor takes parameters.
- **Methods:** `super.methodName()` calls the parent's version of a method (useful if overriding).

Example:

```
Hero(String name, int age, String power) {
    super(name, age); // Passes name/age to Person constructor
    this.power = power;
}
```

Chapter 46: Method Overriding

Timestamp: 7:41:40

Declaring a method in a child class that is already present in the parent class.

Key Points:

- **Annotation:** `@Override` (Good practice for validation).
- **Purpose:** Give a specific implementation for the child class (e.g., `Animal` has generic `speak()` , but `Dog` overrides it to "Bark").

Chapter 47: toString Method

Timestamp: 7:46:09

A special method inherited from the `Object` class.

- **Default Behavior:** Returns the object's memory address (hash code).
- **Overriding:** Commonly overridden to return a meaningful string representation of the object (e.g., field values).

- **Implicit Call:** passing the object to `System.out.println(myObject)` automatically calls `myObject.toString()`.

```
@Override
public String toString() {
    return make + "\n" + model + "\n" + color + "\n" + year;
}
```

Chapter 48: Abstraction

Timestamp: 7:52:03

Abstract Class:

- Cannot be instantiated (cannot do `new Vehicle()`).
- Serves as a strict blueprint for subclasses.
- Keyword: `abstract`.

Abstract Method:

- Declared without implementation (no body).
- **Must** be implemented (overridden) by any concrete child class.
- Example: `abstract void go();` forces all vehicles to define *how* they go.

Chapter 49: Interfaces

Timestamp: 8:01:32

A blueprint for a class that specifies a set of methods that **must** be implemented.

Key Points:

- **Keyword:** `interface` (to define), `implements` (to use).
- **Multiple Inheritance:** A class can implement *multiple* interfaces (unlike extending classes).
- **Contract:** Functions as a contract; if a class implements `Predator`, it *must* define the `hunt()` method.

Example:

```
public class Hawk implements Predator {
    @Override
    public void hunt() {
        System.out.println("The hawk is hunting");
    }
}
```

Chapter 50: Polymorphism

Timestamp: 8:07:45

"Many shapes". The ability of an object to identify as more than one type.

Example:

- A `Corvette` is a `Corvette`, but also a `Car`, a `Vehicle`, and an `Object`.

- **Usage:** You can create an array of `Vehicle[]` to store `Car` , `Boat` , and `Bicycle` objects together because they all share the `Vehicle` parent.

Chapter 51: Dynamic Polymorphism (Runtime)

Timestamp: 8:14:29

The ability to determine which method implementation to execute at **runtime** (while the program is running) based on the actual object type.

Example:

- Declaring `Animal animal;`
- User chooses "Dog" -> `animal = new Dog();`
- Calling `animal.speak()` will execute the *Dog's* speak method, even though the variable type is `Animal` .

Chapter 52: Encapsulation (Getters and Setters)

Timestamp: 8:19:40

Hiding sensitive data (variables) of a class and controlling access via methods.

Steps:

1. **Private:** Make attributes `private` (`private String make;`).
2. **Public Methods:** Create public Setters and Getters.
    - **Getters:** Read-only access (`getMake()`).
    - **Setters:** Write access (`setMake()`). Allowing you to validate data before assignment (e.g., ensure `year` is not negative).

Chapter 53: Aggregation (Partial)

Timestamp: 8:29:39

Concept:

- Represents a **"Has-A"** relationship.
- One object contains another object as part of its structure.
- The contained objects **can exist independently** of the container.

Example:

- **Library** has **Books**.
- If you destroy the Library, the Books still exist.
- *Note: Transcript cuts off during the creation of Book objects.*

Chapter 54: Composition

Timestamp: 8:39:02

Concept:

- Represents a **"Part-Of"** relationship.
- One object is composed of other objects (e.g., A `Car` *has* an `Engine`).

- **Key Difference from Aggregation:** If the parent object is destroyed, the child objects are also destroyed (they don't exist independently).

Example:

```
public class Car {
    private Engine engine; // Composition

    Car(String engineType) {
        this.engine = new Engine(engineType);
    }
}
```

Chapter 55: Wrapper Classes

Timestamp: 8:45:18

Classes that allow primitive data types to be used as objects.

Key Concepts:

- **Autoboxing:** Automatic conversion of primitive to wrapper (e.g., `int` to `Integer`).
- **Unboxing:** Automatic conversion of wrapper to primitive.
- **Usage:** Essential for Collections (like `ArrayList`) which can only store objects, not primitives.

Primitive	Wrapper Class
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Utility Methods: `Integer.parseInt("123")` , `Double.toString(3.14)` .

Chapter 56: ArrayLists

Timestamp: 8:55:52

A resizable array that stores objects. Elements can be added and removed dynamically.

Key Methods:

- `add(element)` : Adds to the end.
- `get(index)` : Retrieves element.
- `set(index, element)` : Replaces element.
- `remove(index)` : Removes element.
- `size()` : Returns number of elements.
- `Collections.sort(list)` : Sorts the list.

Syntax:

```
ArrayList<String> food = new ArrayList<>();
food.add("Pizza");
food.add("Hamburger");
```

Chapter 57: Exception Handling

Timestamp: 9:05:33

Handling runtime errors so the program flow isn't interrupted.

Structure:

- **try:** Surround dangerous code that might throw an error.
- **catch:** Handle specific exceptions (e.g., `ArithmeticException`, `InputMismatchException`).
- **finally:** Block that *always* executes (used for cleanup/closing resources).

Example:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("You can't divide by zero!");
} finally {
    System.out.println("This always runs.");
}
```

Chapter 58: Write Files

Timestamp: 9:13:30

Writing text to files using `FileWriter`.

Key Points:

- Must handle `IOException`.
- **FileWriter:** Arguments include file name/path.
- **write()**: Writes the string.
- **close()**: Crucial to close the writer to save changes.

Example:

```
try {
    FileWriter writer = new FileWriter("poem.txt");
    writer.write("Roses are red\nViolets are blue");
    writer.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Chapter 59: Read Files

Timestamp: 9:22:01

Reading text files using `FileReader` and `BufferedReader`.

Key Points:

- `FileReader`: Reads the file.
- `BufferedReader`: Reads text efficiently (e.g., `readLine()` for line-by-line).
- **Loop:** `while ((line = reader.readLine()) != null)` prints each line until end of file.

Chapter 60: Project - Music Player

Timestamp: 9:28:51

Playing `.wav` audio files using Java's `javax.sound.sampled` library.

Key Components:

- `File`: Point to the `.wav` file.
- `AudioInputStream`: Stream for reading audio data.
- `Clip`: Controls playback (start, stop, reset).
- **Thread/Scanner Block:** The program typically ends immediately after `clip.start()`. We need a loop (like a `Scanner` wait or `while` loop) to keep the program alive while music plays.

Controls Implemented:

- `P` = Play (`clip.start()`)
- `S` = Stop (`clip.stop()`)
- `R` = Reset (`clip.setMicrosecondPosition(0)`)
- `Q` = Quit (`clip.close()`)

Chapter 61: Project - Hangman Game

Timestamp: 9:43:16

A command-line game where players guess a word letter by letter.

Features:

- **Word State:** `ArrayList<Character>` initialized with underscores (`_`).
- **Input Validation:** Checking if input is a letter and if it was already guessed.
- **Logic:**
  - If guess is correct -> Reveal letter in `wordState`.
  - If guess is wrong -> Increment wrong guesses, draw ASCII art.
- **Dictionary:** Reading a random word from a `words.txt` file (using `BufferedReader` and `Random`).

Chapter 62: Dates & Times

Timestamp: 10:11:44

Modern Java Date/Time API (`java.time` package).

Classes:

- `LocalDate`: Date only (yyyy-MM-dd).
- `LocalTime`: Time only (HH:mm:ss).
- `LocalDateTime`: Both date and time.

- `DateTimeFormatter` : Custom formatting (e.g., "MM/dd/yyyy").

Example:

```
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
String formattedDate = now.format(formatter);
```

Chapter 63: Anonymous Classes

Timestamp: 10:20:28

A class without a name, defined and instantiated in a single expression.

Usage:

- Useful for one-time use (e.g., overriding a method of an object on the fly).
- Avoids creating a separate `.java` file for a class used only once.

Example:

```
Dog myDog = new Dog() {
    @Override
    void speak() {
        System.out.println("This specific dog speaks English!");
    }
};
```

Chapter 64: TimerTask

Timestamp: 10:25:27

Scheduling tasks to run at a specific time or repeatedly.

Components:

- `Timer` : The scheduler.
- `TimerTask` : The task to run (implements `run()`).
- `timer.schedule(task, delay)` : Run once after delay.
- `timer.scheduleAtFixedRate(task, delay, period)` : Run repeatedly.

Chapter 65: Project - Countdown Timer (Partial)

Timestamp: 10:31:49

Using `Timer` and `TimerTask` to create a countdown.

Logic:

- Inside `TimerTask.run()` :
    - Print current number.
    - Decrement counter.
    - If counter < 0, print "Happy New Year!" and call `timer.cancel()` .

- Note: The transcript cuts off during the setup of the user input for the timer.

Chapter 65: Project - Countdown Timer (Completed)

Timestamp: 10:34:37

Logic (Continued):

1. **Scheduling:** Used `timer.scheduleAtFixedRate(task, 0, 1000)` to execute the task every 1000ms (1 second).
2. **User Input:** Added a `Scanner` to let the user input the starting number of seconds.
3. **Cancellation:** Crucial to call `timer.cancel()` inside the `run()` method when the counter reaches 0 to stop the program; otherwise, it runs forever.

Chapter 66: Generics

Timestamp: 10:38:21

Enables classes, interfaces, and methods to take valid types as parameters, providing compile-time type safety.

Key Concepts:

- **Type Parameters:** `<T>` acts as a placeholder for a type.
- **Diamond Operator:** `<>` inferred type (e.g., `new ArrayList<>`).
- **Code Reusability:** One class can handle Strings, Integers, etc., without code duplication.

Example 1: Generic Class

```java
public class Box<T> {
    T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return this.item;
    }
}
// Usage:
Box<String> box = new Box<>();
box.setItem("Pizza");
```

Example 2: Multiple Parameters

```java
public class Product<T, U> {
    T item;
    U price;
    // ... constructor and getters
}
// Usage:
Product<String, Double> p = new Product<>("Apple", 0.50);
```

Chapter 67: HashMaps

Timestamp: 10:52:10

A data structure that stores items in **Key-Value** pairs.

- **Keys:** Must be unique. Adding a duplicate key overwrites the old value.
- **Values:** Can be duplicates.
- **Order:** No guaranteed order.

Syntax: `HashMap<String, Double> map = new HashMap<>();`

Key Methods:

- `put(key, value)` : Insert items.
- `get(key)` : Retrieve value.
- `remove(key)` : Remove pair.
- `containsKey(key)` : Checks if key exists.
- `size()` : Number of pairs.
- `keySet()` : Returns set of keys (useful for iteration).

Iterating:

```
for (String key : map.keySet()) {
    System.out.println(key + " : " + map.get(key));
}
```

Chapter 68: Enums

Timestamp: 11:02:41

Enumerations are a special kind of class that represents a fixed set of constants.

Key Benefits:

- Type safety.
- More readable than integers or strings.
- Efficient in `switch` statements.

Implementation:

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Advanced Enums: Enums can have fields, constructors, and methods.

```
enum Day {
    SUNDAY(1), MONDAY(2); // ...

    final int dayNumber;

    Day(int dayNumber) {
        this.dayNumber = dayNumber;
    }
}
```

Chapter 69: Threading

Timestamp: 11:12:45

Allows a program to run multiple tasks simultaneously (concurrently). Useful for background tasks or time-consuming operations so the main program doesn't freeze.

Methods to Create Threads:

1. **Extend `Thread` class.**
2. **Implement `Runnable` interface** (Preferred, allows implementing other interfaces).

Key Concepts:

- `run()` : The code that executes in the new thread.
- `start()` : Begins execution of the thread (calls `run()` internally).
- `Thread.sleep(millis)` : Pauses execution.
- `setDaemon(true)` : Daemon threads run in the background and terminate automatically when the main thread finishes.
- `join()` : Waits for a thread to die (finish) before proceeding.

Example (Runnable):

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
thread.start();
```

Chapter 70: Multithreading

Timestamp: 11:23:05

Running multiple threads concurrently.

Example:

- Two threads counting to 5 simultaneously.
- Output order is not guaranteed (they run independently).
- **Anonymous Inner Class Shortcut:**

  ```
  Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
          // task code
      }
  });
  ```

Chapter 71: Final Project - Alarm Clock

Timestamp: 11:31:05 A comprehensive project combining Date/Time, Threading, and Audio.

Architecture:

- **Main Class:** Handles user input (`Scanner`) to set the `LocalTime` for the alarm.
- **AlarmClock Class:** Implements `Runnable` . Runs on a separate thread.
    - **Loop:** Continually checks `LocalTime.now()` .
    - **Logic:** `if (now.isBefore(alarmTime))` , sleep for 1 second. Else, play audio.
    - **Audio:** Uses `Clip` to play a `.wav` file (similar to the Music Player project).
    - **Formatting:** Uses `printf` with `\r` (carriage return) to update the time display in place without flooding the console.
    - **Stop Condition:** Waits for user to press Enter in the main console to call `clip.stop()` .

Key Code Snippet (Time Check Loop):

```java
while (LocalTime.now().isBefore(alarmTime)) {
    try {
        Thread.sleep(1000);
        LocalTime now = LocalTime.now();
        System.out.printf("\r%02d:%02d:%02d", now.getHour(), now.getMinute(),
now.getSecond());
    } catch (InterruptedException e) {
        // handle exception
    }
}
playSound("alarm.wav");
```