**Week 1**

**Chapter 1**

**Introduction to Computers and Java programming Language**

**Day 1**

**Installation**

**Introduction**

Java is one of the world’s most widely used computer programming languages. It was created in 1995 by a team led by James Gosling at Sun Microsystems, later acquired by Oracle Corporation and now it is owned by Oracle and more than 3 billion devices run Java.

**Some devices that run java**

Airplane systems

ATMs

Automobile infotainment systems

Blu-ray Disc™ players

Cable boxes

Copiers

Credit cards

CT scanners

Desktop computers

e-Readers

Game consoles

GPS navigation systems

Home appliances

Home security systems

Light switches

Lottery terminals

Medical devices

Mobile phones

MRIs

Parking payment stations

Printers

Transportation passes

Robots

Routers

Smart cards

Smart meters

Smart pens

Smartphones

Tablets

Televisions

TV set-top boxes

Thermostats

Vehicle diagnostic systems

Projectors

**What Java is used for**

Java is a powerful and versatile programming language used in many areas, including:

1. **Web Development** – Used for building websites and web applications with frameworks like Spring and Jakarta EE.
2. **Mobile Apps** – A key language for Android app development, though Kotlin is now preferred.
3. **Enterprise Applications** – Used for large business systems like CRM (Customer Relationship Management) and ERP (Enterprise Resource Planning) software.
4. **Desktop Applications** – Can create cross-platform desktop apps using JavaFX and Swing.
5. **Big Data** – Supports big data tools like Apache Hadoop and Spark.
6. **Scientific Computing** – Used in research, AI, and natural language processing.
7. **Embedded Systems** – Powers IoT devices, smart cards, and sensors.
8. **Cloud Computing** – Supports cloud-based applications on AWS, Google Cloud, and Azure.
9. **Gaming** – Used in game development, including Minecraft and LibGDX-based games.
10. **Finance & Banking** – Essential for secure, high-performance trading and banking software.
11. **Automation** – Helps create scripts and tools for system management and workflow automation.
12. **Machine Learning & AI** – Has libraries for AI and data science, though Python is more common.
13. **IoT (Internet of Things)** – Supports IoT platforms for scalable, connected devices.

Java is widely used because it is reliable, scalable, and works across different platforms.

**Java Terminology**

Before learning Java, one must be familiar with these common terms of Java.

### ****Java Development Kit (JDK)****

The **Java Development Kit (JDK)** is a tool for building Java applications. It includes everything you need to write, compile, and run Java programs.

**Key Parts of JDK:**

1. **Java Compiler (javac)** – Converts Java code into bytecode so it can run on any system.
2. **Java Runtime Environment (JRE)** – Helps Java programs run on your computer. It includes:
   * **Java Virtual Machine (JVM)** – Translates and runs Java bytecode on any device, making Java platform-independent.
   * **Libraries & Files** – Pre-written code that Java programs use for common tasks.
3. **Development Tools** – Useful tools like:
   * **javadoc** (creates documentation)
   * **jdb** (debugging Java programs)
   * **javap** (disassembles bytecode)

**Note:** If you want to develop Java programs, you **must** install the JDK.

### ****How Java Code Runs – There are 3 Main Phases****

1. **Writing Phase** – A programmer writes Java code.
2. **Compilation Phase** – The **javac** compiler converts the code into bytecode.
3. **Running Phase** – The **JVM** runs the bytecode on your computer.

### ****Garbage Collection – Automatic Memory Cleanup****

Think of garbage collection like cleaning a room—removing things you no longer need to keep it tidy.

* When a program runs, it uses memory to store data.
* Over time, some data is no longer needed.
* The **garbage collector** automatically removes unused data, freeing up memory.

This process happens in the background, so developers don’t need to manually delete old data. It keeps Java programs running smoothly.

**Main Features of Java**

1. **Platform Independent** – Java code is compiled into **bytecode**, which can run on any operating system (Windows, Linux, macOS) using the Java Virtual Machine (**JVM**).
2. **Object-Oriented Programming (OOP)** – Java is based on OOP principles, which help organize code using **objects and classes**. The four main OOP concepts are:
   * **Abstraction** (Hiding complex details)
   * **Encapsulation** (Protecting data inside a class)
   * **Inheritance** (Reusing code from other classes)
   * **Polymorphism** (Using the same method in different ways)
3. **Simple** – Java avoids complex features like **pointers**, **operator overloading**, and **manual memory management**, making it easier to learn.
4. **Robust (Reliable)** – Java helps prevent errors with **automatic garbage collection**, **exception handling**, and **strong memory management**.
5. **Secure** – Java has built-in security features to prevent **memory leaks, unauthorized access, and errors** that could crash programs.
6. **Distributed** – Java supports creating applications that work over the **internet** and across multiple computers using **Remote Method Invocation (RMI)** and **Enterprise Java Beans (EJB)**.
7. **Multithreading** – Java can run multiple tasks **at the same time**, making programs **faster and more efficient**.
8. **Portable** – Java programs can run on **any device or OS** without modification, thanks to its **platform-independent bytecode**.
9. **High Performance** – Java uses a **Just-In-Time (JIT) compiler**, which **compiles code only when needed**, improving speed and efficiency.
10. **Dynamic & Flexible** – Java allows **adding new features** to existing programs and supports **calling code written in other languages** like C and C++.
11. **Write Once, Run Anywhere** – Java programs don’t depend on a specific **computer or architecture**, making them widely used in the **IT industry worldwide**.

In short, **Java is fast, secure, portable, and easy to use**, making it one of the most popular programming languages today!

**Editions of Java**

**There are four types of Java edition:**

1. Java Standard Edition (Java SE)
2. Java Enterprise Edition (Java EE)
3. Java Micro Edition (Java ME).
4. JavaFX

**1. Java Standard Edition (Java SE) – Core Java**

Java SE is the foundation of Java and is used to develop **desktop, server, and general applications**.

**Key Features:**

* **Core APIs** – Basic libraries for input/output, networking, and utilities.
* **Multithreading** – Runs multiple tasks at once.
* **Java Collections Framework** – Provides data structures like lists and maps.
* **JavaFX (UI)** – Used for building desktop applications.
* **Functional Programming** – Features like Lambda Expressions (from Java 8).

**Used for:**  
1. Desktop apps (e.g., Notepad-like applications)  
2. Server-side applications (e.g., APIs and backend services)  
3. Console-based applications

**2. Java Enterprise Edition (Java EE) (Now Jakarta EE) – Enterprise Java**

Java EE extends Java SE and is used for **large-scale web applications and enterprise systems**.

**Key Features:**

* **Web Development** – Supports **Servlets, JSP, and RESTful APIs**.
* **Database Access** – Uses **JPA (Java Persistence API)** for database connections.
* **Messaging** – **JMS (Java Message Service)** for communication between apps.
* **Security** – Provides authentication and authorization features.
* **Transaction Management** – **JTA (Java Transaction API)** for handling complex transactions.

**Used for:**  
1. Large enterprise apps (e.g., banking systems, ERP, CRM)  
2. Web applications (e.g., online shopping websites)  
3. Business applications requiring messaging and database access

🔹 **Note:** Java EE is now called **Jakarta EE** and is maintained by the Eclipse Foundation.

**3. Java Micro Edition (Java ME) – Java for Small Devices**

Java ME is a **lightweight version** of Java SE designed for **mobile and embedded systems**.

**Key Features:**

* **Optimized for small devices** – Uses **CLDC (Connected Limited Device Configuration)** for limited memory.
* **Mobile and IoT Development** – Uses **MIDP (Mobile Information Device Profile)** for small gadgets.
* **Works on Embedded Systems** – Used in **smartwatches, sensors, home appliances, and IoT devices**.

**Used for:**  
1. Older mobile phones and feature phones  
2. Embedded systems (e.g., medical devices, printers)  
3. IoT (Internet of Things) applications

**4. JavaFX – Java for Modern UI**

JavaFX is a **framework for building modern desktop applications with graphical user interfaces (GUIs)**.

**Key Features:**

* **Scene Graph** – Creates **interactive UI components** like buttons, charts, and animations.
* **FXML** – Uses **markup language** for designing UI (similar to HTML).
* **CSS Styling** – Supports **CSS** for custom UI design.
* **3D Graphics** – Can create 3D effects and animations.
* **Multimedia Support** – Plays **audio, video, and animations**.
* **Cross-platform** – Runs on **Windows, macOS, and Linux**.

**Used for:**  
1. Desktop applications with **rich user interfaces**

2. Data visualization tools and **charts**  
3. Media players, enterprise applications

**Types of Programming Languages**

Programmers use different types of languages to write programs. These languages can be divided into three main categories:

**1. Machine Language (Lowest Level - 1s & 0s)**

* Directly understood by a computer.
* Made up of **binary code (1s and 0s)**.
* **Machine-dependent** (works only on one type of computer).
* Hard for humans to read and write.

**2. Assembly Language (Uses Simple Words & Symbols)**

* Uses **English-like abbreviations** instead of binary code.
* Easier to write than machine language.
* Needs a **translator program (Assembler)** to convert it into machine language.

**3. High-Level Language (Easy to Read & Write)**

* Uses **English-like statements and math symbols**.
* **Much simpler** than machine or assembly language.
* Needs a **compiler or interpreter** to convert into machine language.

**Compilers vs. Interpreters**

* **Compiler** – Translates the entire program **before execution** (faster but takes time to compile).
* **Interpreter** – Translates **line by line** while running the program (slower but avoids compilation delay).

Java uses both:

* **Java Compiler (javac)** converts code into **bytecode**.
* **Java Interpreter (JVM)** runs the bytecode on any system.

**Keywords**

**Keywords are predefined words** in Java that have special meanings and cannot be used for variable names or other purposes.

**Examples:** class, public, static, if, else, while, return, void, etc. There are **57** keywords in Java.

**Chapter 2**

### ****Classes and Methods****

### ****1. Declaring a Class****

* Every Java program has at least **one class**.
* Use the **class** keyword to define a class.
* Class names should **start with a capital letter** and match the filename (ClassName.java).

**Example:**

public class MyClass {

// Class body goes here

}

### ****2. Class Names and Identifiers****

**Class names** must follow these rules:  
Start with a **capital letter** (e.g., MyClass, Welcome1).  
Can contain **letters, numbers, \_, and $**.  
Cannot start with a **number** or contain **spaces**.

* **Java is case-sensitive**, so value and Value are different.

🔹 **Valid Identifiers:** Welcome1, $value, \_value, button7  
🔹 **Invalid Identifiers:** 7button (starts with a digit), input field (contains a space)

### ****3. Class Body**** { }

* **Curly braces { }** define where a class starts and ends.
* **Example:**

java

Copy

public class Example {

// Class body starts

} // Class body ends

### ****4. Main Method (Starting Point of Every Java Program)****

* **Java programs start execution from the main method**.
* **Syntax:**

java

Copy

public static void main(String[] args) {

// Code to execute

}

**Day 2**

**Output Using print(), println(), and printf() methods**

In Java, you can use System.out.print(), System.out.println(), and System.out.printf() methods to output text to the console. Each method serves different purposes for formatting and displaying data.

**System.out.print():** Outputs text to the console without appending a newline character at the end.

**System.out.println():** Outputs text to the console and appends a newline character at the end, which moves the cursor to the next line.

**Escape characters**

Escape characters are used within string literals to represent special characters or to include characters that would otherwise be difficult to include directly in the string

|  |  |
| --- | --- |
| **Escape Characters** | **Description** |
| **\t** | It is used to insert a **tab** in the text at this point. |
| **\'** | It is used to insert a **single quote** character in the text at this point. |
| **\"** | It is used to insert a **double quote** character in the text at this point. |
| **\r** | It is used to insert a **carriage return** in the text at this point. |
| **\\** | It is used to insert a **backslash character** in the text at this point. |
| **\n** | It is used to insert a **new line** in the text at this point. |
| **\f** | It is used to insert a **form feed** in the text at this point. |
| **\b** | It is used to insert a **backspace** in the text at this point. |

**System.out.printf():** Outputs text with formatted data. It allows you to format numbers, strings, and other data types with specific formatting options.Here are some common format specifiers you can use with printf():

* **%s**: String
* **%d**: Decimal integer
* **%f**: Floating-point number
* **%.nf**: Floating-point number with n decimal places
* **%c**: Character
* **%b**: Boolean
* **%n**: Newline

**Using Blank Lines**

Blank lines, space characters and tabs make programs easier to read. Together, they’re known as white space (or whitespace). The compiler ignores white space.

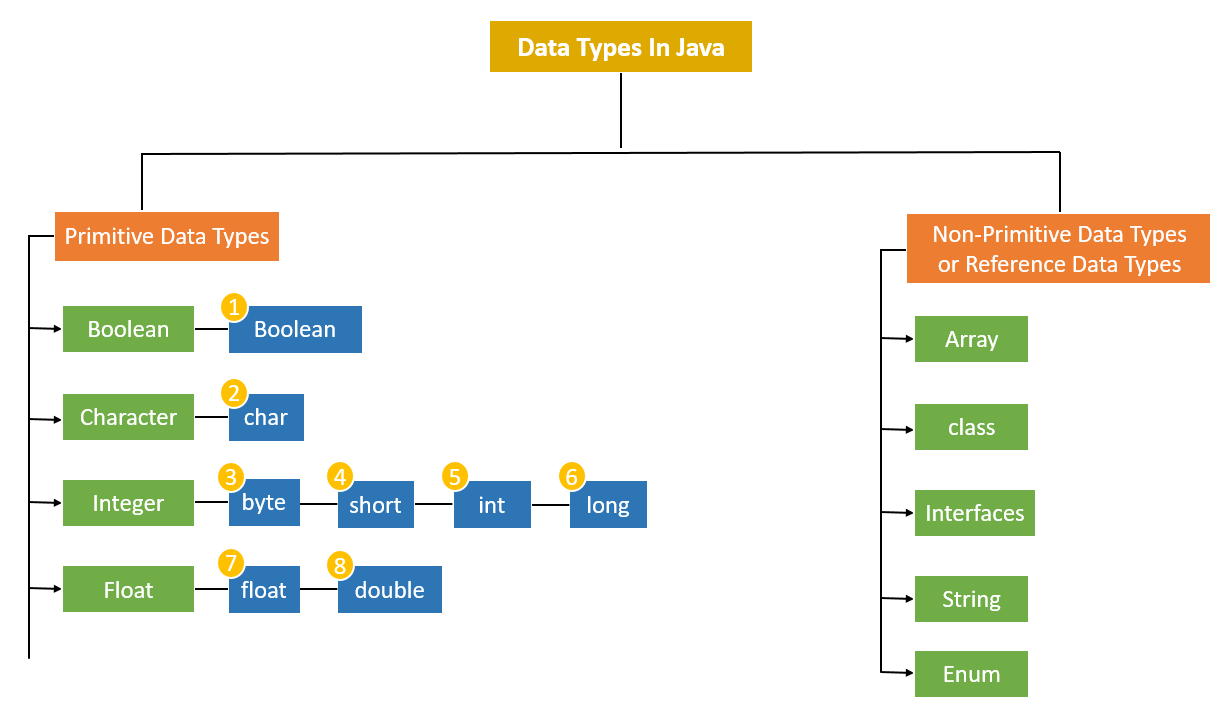
**Data types**

**Data types refer to the different sizes and values that can be stored in the variable.** In Java datatypes are like different kinds of containers that hold different types of data.

**In java there are two types of datatypes**

They are:

* Primitive datatype
* Non-primitive datatype or reference



**Primitive Data Types**

A primitive data type is pre-defined by the programming language. The size and type of variable values are specified, and it has no additional methods**.** They are the most basic data types available in Java. They directly hold exactly one value of its declared type at a time and are not objects.

**Primitive data are stored on the stack. Primitive** data types in Java are classified into 4 aspects as**Character**, **Boolean, Integer and Float.** But, in general, there are 8 data types

1. **boolean**: Represents a true or false value.
   * Default value: false
   * Example: boolean isJavaFun = true;
2. **char**: Represents a single 16-bit Unicode character.
   * Default value: '\u0000' (null character)
   * Example: char myChar = 'A';
3. **byte**: Represents a 16-bit signed integer.
   * Range: 0-255
   * Default value: 0
   * Example: short myByte = 50;
4. **short**: Represents a 16-bit signed integer.
   * Range: -32,768 to 32,767
   * Default value: 0
   * Example: short myShort = 500;
5. **int**: Represents a 32-bit signed integer.
   * Range: -2^31 to 2^31 - 1
   * Default value: 0
   * Example: int myInt = 100000;
6. **long**: Represents a 64-bit signed integer.
   * Range: -2^63 to 2^63 - 1
   * Default value: 0L (note the L suffix)

In Java, when you specify a literal for a long data type, you must append an L (or l, though L is preferred for clarity) to the end of the number. This suffix indicates to the Java compiler that the literal should be treated as a long value rather than an int.

Here's why this distinction is important:

* **Default Interpretation**: Without the L suffix, Java treats integer literals as int by default. The int data type can only hold values within the range of approximately -2 billion to +2 billion. If you try to assign a number larger than this range to a variable without the L suffix, you'll get a compilation error because it exceeds the limits of int.
* **Explicit Type Declaration**: By appending L to a number (e.g., 10000000000L), you explicitly tell Java that this number should be treated as a long. A long can hold much larger values than int, ranging from approximately -9 quintillion to +9 quintillion.

1. **float**: Represents a 32-bit floating point number.
   * Default value: 0.0f (note the f suffix)

In Java, when you specify a literal for a float data type, you must append an f (or F, though f is commonly used for clarity) to the end of the number. This suffix indicates to the Java compiler that the literal should be treated as a float value rather than a double.

Here's why this distinction is important:

* **Default Interpretation**: Without the f suffix, Java treats floating-point literals as double by default. double can hold larger and more precise values compared to float, but it requires more memory.
* **Explicit Type Declaration**: By appending f to a number (e.g., 3.14f), you explicitly tell Java that this number should be treated as a float. This is necessary because float and double are different in terms of precision and memory usage.

1. **double**: Represents a 64-bit floating point number.
   * Default value: 0.0d (note the d suffix, though it's optional)
   * Example: double myDouble = 3.14159;

**Non-Primitive Data Types:**

These data types are not actually defined by the programming language but are created by the programmer. They are also called “reference variables” or “object references” since they reference a memory location which stores the data. Non primitive data are store on the heap but the stack hold the pointer to the object in the heap.

1. **String**: Represents a sequence of characters. In Java, strings are objects of the String class. Eg String str = "Hello, Java!";
2. **Arrays**: Collections of similar types of data. Arrays in Java are objects with a fixed number of elements. Eg. int[] numbers = {1, 2, 3, 4, 5};
3. **Classes**: User-defined data types that encapsulate data and methods. Instances of classes are objects.

class Person {

String name;

int age;

}

Person person1 = new Person();

person1.name = "Alice";

person1.age = 30;

1. **Interfaces**: Defines a contract that implementing classes must follow. An interface in Java is also a reference type.

interface Shape {

void draw();

}

class Circle implements Shape {

@Override

public void draw() {

// draw circle logic

}

}

Shape shape = new Circle();

1. **Enums**: Special data types that allow for a variable to be a set of predefined constants.

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

Day today = Day.MONDAY;

1. **Wrapper Classes**: Java provides wrapper classes to wrap primitive data types in an object. These are used when an object is required instead of a primitive type.

Integer num = new Integer(10); // Wrapper class for int

Double price = new Double(19.99); // Wrapper class for double

1. **Arrays of Objects**: Array of Objects is used to stores an array of objects. Unlike the traditional array stores values like String, integer, Boolean, etc. an Array of Objects stores **objects,**that means objects are stored as elements of an array. Note that when we say **Array of Objects** it is not the object itself that is stored in the array but the reference of the object. Arrays can also hold references to objects. For example:

Person[] people = new Person[5]; // Array of Person objects

people[0] = new Person("Bob", 25);

**Type Casting for primitive type**

Type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

**Types of Type Casting**

There are two types of type casting:

* Widening Type Casting
* Narrowing Type Casting



**Widening Type Casting**

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

* Both data types must be compatible with each other.
* The target type must be larger than the source type.

**byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

**Note:** the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other.

**Narrowing Type Casting**

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

**double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

**Tokens**

In Java, **tokens** refer to the smallest units of meaningful data that a Java compiler recognizes in a program's source code. These tokens are the building blocks for Java programs. There are several types of tokens in Java, which can be classified as follows:

### 1. ****Keywords****

* Keywords are reserved words in Java that have predefined meanings and cannot be used as identifiers (e.g., variable names or class names).
* Examples of keywords include:
  + class, public, private, if, else, int, for, while, return, break, static, etc.

### 2. ****Identifiers****

* Identifiers are names given to variables, methods, classes, and other user-defined elements in a Java program.
* An identifier must begin with a letter (a-z or A-Z), a dollar sign ($), or an underscore (\_). After the first character, it can also contain digits (0-9).
* Examples of valid identifiers:
  + myVariable, sum, main, MyClass, \_counter, total$

### 3. ****Literals****

* Literals are constant values used in Java programs, such as numeric values, characters, and strings.
* Examples include:
  + Numeric literals: 10, 3.14, 100L
  + String literals: "Hello, world!"
  + Character literals: 'A', '1'
  + Boolean literals: true, false

### 4. ****Operators****

* Operators are symbols used to perform operations on variables and values. Java has several types of operators, including:
  + **Arithmetic operators**: +, -, \*, /, %
  + **Relational operators**: ==, !=, >, <, >=, <=
  + **Logical operators**: &&, ||, !
  + **Assignment operators**: =, +=, -=, \*=, /=
  + **Unary operators**: ++, --, +, -

### 5. ****Separators (Punctuation)****

* Separators help organize Java code by defining structure and grouping code elements together.
* Common separators in Java include:
  + **Semicolon (;)**: Marks the end of a statement.
  + **Comma (,)**: Separates arguments in a method call or items in a list.
  + **Parentheses (())**: Used in method calls, loops, conditionals, etc.
  + **Curly braces ({})**: Define the body of classes, methods, loops, etc.
  + **Square brackets ([])**: Used for arrays.
  + **Dot (.)**: Used to access members of a class or an object (e.g., System.out.println()).

### 6. ****Comments****

* Comments are used for documentation and to make the code more understandable. They are ignored by the compiler.
  + **Single-line comments**: // This is a comment
  + **Multi-line comments**: /\* This is a multi-line comment \*/
  + **Javadoc comments**: /\*\* This is a Javadoc comment \*/

**Day 3**

**Getting Input from User**

Java **Scanner class** allows the user to take input from the console. It belongs to **java.util** package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in Java program.

**Using Scanner to Read Strings**

1. **next() Method:** Reads the next token from the input. By default, tokens are delimited by whitespace (spaces, tabs, newlines).
2. **nextLine() Method:** Reads the entire line of input until the end of the line is encountered. This is useful when you want to read a full line of text including spaces.
3. **Extract the Character** Convert the input string to a character using the charAt() method.

**Integer Methods**

1. **nextByte()** **:** Reads the next token of input as a byte.
2. **nextShort()** **:** Reads the next token of input as a short.
3. **nextInt():** Reads the next token of input as an int.
4. **nextLong():** Reads the next token of input as a long.

**Floating-Point Methods**

1. **nextFloat():** Reads the next token of input as a float.
2. **nextDouble():** Reads the next token of input as a double.

**Boolean Method**

* **nextBoolean():** Reads the next token of input as a boolean. The token must be either "true" or "false", ignoring case.
* Be cautious when using nextBoolean() after other input methods (like nextInt() or nextLine()). If you mix different types of input, you might need to call scanner.nextLine() to consume the leftover newline character.

**Notes**

* Make sure to handle exceptions, especially when reading numeric input. If the user inputs a non-numeric value when nextInt(), nextDouble(), or similar methods are called, it will throw an InputMismatchException and NoSuchElementException if no more tokens are available.
* Be aware that nextLine() can sometimes be problematic after using nextInt(), nextDouble(), etc., because it may consume the newline character left in the buffer. It’s often recommended to call nextLine() after reading other types of input to clear the buffer.

**Operators**

Operators are special symbols that perform operations on variables and values. Java has a rich set of operators categorized into different types.

* 1. **Assignment Operators**
* **Simple Assignment (=)**: Assigns a value to a variable.
* **Add and Assign (+=)**: Adds a value to the variable and assigns the result.
* **Subtract and Assign (-=)**: Subtracts a value from the variable and assigns the result.
* **Multiply and Assign (\*=)**: Multiplies the variable by a value and assigns the result.
* **Divide and Assign (/=)**: Divides the variable by a value and assigns the result.
* **Modulus and Assign (%=)**: Applies modulus operation and assigns the result.
  1. **Arithmetic Operators:** Used for basic mathematical operations.
* **Addition (+)**: Adds two values.
* **Subtraction (-)**: Subtracts the second value from the first.
* **Multiplication (\*)**: Multiplies two values.
* **Division (/)**: Divides the first value by the second.
* **Modulus (%)**: Returns the remainder of the division.
  1. **Comparison or Relational Operators:** Used to compare two values.
* **Equal to (==)**: Checks if two values are equal.
* **Not equal to (!=)**: Checks if two values are not equal.
* **Greater than (>)**: Checks if the first value is greater than the second.
* **Less than (<)**: Checks if the first value is less than the second.
* **Greater than or equal to (>=)**: Checks if the first value is greater than or equal to the second.
* **Less than or equal to (<=)**: Checks if the first value is less than or equal to the second.
  1. **Logical Operators:** Used to perform logical operations.
* **Logical AND (&&)**: Returns true if both conditions are true.
* **Logical OR (||)**: Returns true if at least one condition is true.
* **Logical NOT (!)**: Reverses the logical state.

**Chapter 3**

**Day 4**

**Control structures**

control structures are used to manage the flow of execution based on certain conditions or loops. There are three primary types of control structures:

1. selection (decision),
2. repetition (loop),
3. and jump.

**selection statements**

selection statements are used to make decisions based on conditions. There are Four(4) main types:

1. **if statement**: A single-selection statement performs an action if a condition is true, and skips it if false.
2. **if…else statement**: A double-selection statement performs one action if the condition is true and a different action if the condition is false.
3. **nested if...else** statement allows you to test multiple conditions by placing an if...else statement inside another if...else statement. This structure helps handle more complex decision-making in a program.
4. **switch statement**: A multiple-selection statement that performs one of many actions depending on the value of an expression.

**Conditional Operator**

the conditional operator (also known as the ternary operator) in Java is a shorthand way to write simple if...else statements. It helps make your code more concise and readable.

Syntax of the Conditional Operator (?:)

condition ? value\_if\_true : value\_if\_false;

Condition: A boolean expression that is evaluated (true or false).

value\_if\_true: The value returned if the condition is true.

value\_if\_false: The value returned if the condition is false.

**Week 2**

**Chapter 4**

**Day 1**

**Repetition (Loop)**

Repetition in programming refers to the process of executing a set of instructions or a block of code repeatedly based on a condition or a specific number of times. Loops are used to automate repetitive tasks, allowing the same code to be executed multiple times without needing to rewrite it.

In Java, there are three primary types of repetition statements (loops):

* 1. for loop
  2. while loop
  3. do … while loop

### ****For Loop****

A for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

#### **Syntax:**

for (initialization; condition; increment/decrement) {

// body of loop

}

### 2. ****While Loop****

A while loop is used when the condition is checked before executing the loop's body. The loop continues as long as the condition evaluates to true and terminate when the condition evaluates to false.

#### **Syntax:**

while (condition) {

// body of loop

Increment/decrement

}

### 3. ****Do-While Loop****

A do…while loop is similar to a while loop, except that the condition is evaluated **after** the loop’s body is executed, ensuring that the loop runs at least once.

#### **Syntax:**

do {

// body of loop

} while (condition);

**Local Variable**

A **local variable** in Java is a variable that is declared within a **method, constructor, or block** and can only be accessed within that method, constructor, or block.

**Key Characteristics of Local Variables:**

1. **Declared inside methods, constructors, or blocks**: Local variables are declared within a method or block and cannot be accessed outside of it.
2. **Scope**: The scope of a local variable is limited to the method or block in which it is declared. It cannot be accessed by other methods or classes.
3. **Lifetime**: Local variables are created when the method or block is called and are destroyed when the method or block finishes executing.
4. **No default value**: Unlike instance or class variables, local variables in Java do not have a default value. They must be explicitly initialized before use.
5. **Cannot be static**: Local variables cannot be declared as static.

**Day 2**

**Sentinel controlled repetition**

A **sentinel loop** is a loop that uses a special value (known as the sentinel value) to signal the end of input or to terminate the loop. The sentinel value is typically a value that is not part of the normal data set, used specifically to indicate the termination condition of the loop.

The sentinel value is often used in scenarios where the exact number of iterations is unknown, and the loop should continue until the sentinel value is encountered. This kind of loop is common when reading data or interacting with users, such as in situations where you don't know how many pieces of input, you'll receive but need a way to stop when input reaches a certain marker (the sentinel).

**Increment / decrement operators**

**Increment** and **decrement operators** are used to increase or decrease the value of a variable by 1. These operators provide a shorthand way of modifying the value of a variable in your code.

### 1. ****Increment Operator (++)****

The increment operator increases the value of a variable by 1. It can be used in two forms:

* **Pre-increment** (++variable): Increments the value first, then returns the new value.
* **Post-increment** (variable++): Returns the current value first, then increments the value.

### ****Decrement Operator (--)****

The decrement operator decreases the value of a variable by 1. It can also be used in two forms:

* **Pre-decrement** (--variable): Decreases the value first, then returns the new value.
* **Post-decrement** (variable--): Returns the current value first, then decreases the value.

**Break and continue statements**

**Jump Statements :** They are used to alter the flow of control in loops. There are two types in Java:

* + break statement
  + continue statement

**a. break:** it is used to exits the innermost loop or switch statement.

while (true) {

if (someCondition) {

break; // exit the loop

}

}

**b. continue:** Skips the current iteration of a loop and proceeds with the next iteration.

for (int i = 0; i < 10; i++) {

if (someCondition) {

continue; // skip the rest of this iteration

}

// code to be executed

}

### Assignment: ****Compound Interest Calculation****

#### **Objective:**

Write a Java program that calculates the amount of money in a bank account after 10 years, based on a given initial deposit and annual interest rate. The program should display the amount on deposit at the end of each year, starting from year 1 through year 10.

#### **Problem Description:**

You are tasked with calculating the compound interest for an initial principal amount over a period of 10 years. The interest is compounded annually, and you will display the amount on deposit at the end of each year. You should use the following information:

* **Principal**: The initial amount of money deposited in the account (e.g., $1000.0).
* **Interest Rate**: The annual interest rate expressed as a decimal (e.g., 5% = 0.05).
* **Number of Years**: The program should calculate the deposit for the first 10 years.

**Week 2**

**Chapter 5**

**Day 3**

**Declaring and calling methods**

Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called **divide and conquer.**

**Divide and Conquer: I**t the idea of breaking down a complex program into smaller, more manageable parts or modules. This approach makes it easier to design, implement, test, and maintain large programs.By splitting a program into smaller units (called methods), developers can focus on one piece at a time, improving code organization and reducing complexity.

In Java, methods are blocks of code that perform specific tasks and can be executed when called. Here's how you can declare and call methods in Java.

**1. Declaring a Method**

To declare a method, you need to specify:

* The **access modifier** (e.g., public, private, etc.)
* The **return type** (e.g., int, void, etc.)
* The **method name**
* **Parameters** (optional) inside parentheses
* The method **body**, which contains the code to be executed.

Syntax for creating a method

returnType methodName(parameters)

{ // method body }

### ****Calling a Method****

To call a method, you need to:

* Use the method's name.
* If it requires parameters, pass the appropriate values.
* If the method has a return type, use the returned value.

Syntax for calling:

methodName(arguments);

**Static methods**

In Java, **static methods** are methods that belong to the class rather than instances of the class. This means that you can call a static method without creating an instance of the class. Static methods are used for operations that are common to all objects of the class or when you don't need access to instance variables.

**Key Features of Static Methods:**

1. **Belongs to the class**: A static method is associated with the class itself, not any instance of the class.
2. **Can be called without an instance**: Static methods can be called directly using the class name.
3. **Cannot access instance variables directly**: Since static methods are not tied to any specific object, they cannot directly access instance variables or non-static methods.
4. **Can only access static variables/methods**: Static methods can only access other static variables or static methods directly. To access instance members, you need an instance of the class.

**Syntax for Declaring a Static Method:**

public static returnType methodName(parameters) {

// method body

}

**Hierarchical Relationship Between Method Calls**

The analogy of hierarchical method calls in Java, as described, can be illustrated in terms of method invocation, where a caller (often called a "boss") calls a method (the "worker") to perform a task. The worker method performs the task and returns control (and possibly a result) to the calling method. This structure is a common pattern in programming, particularly in object-oriented programming (OOP) like Java.

Let's break down the key concepts and provide an example in Java based on the analogy described:

**Key Concepts:**

1. **Caller (Boss Method)**: This is the method that invokes other methods. It may call several other methods to divide and conquer the tasks.
2. **Worker Method**: These are the methods called by the caller. They perform a specific task and may also call other methods (further workers).
3. **Task Decomposition**: The caller delegates responsibility to worker methods, which can further delegate to other workers. Each worker is responsible for a specific part of the task

**Math Class Method**

The **Math** class in Java provides a wide variety of static methods for performing basic mathematical operations. These methods are invoked directly using the class name Math followed by a dot (.) and the method name. Below is a list of commonly used static methods in the **Math** class, along with a brief description of each:

1. **Math.abs(double a)**
   * Returns the absolute value of a number.
   * Example: Math.abs(-5.5) returns 5.5.
2. **Math.round(double a)**
   * Returns the closest long to the argument, rounding up if the fractional part is 0.5 or greater.
   * Example: Math.round(4.5) returns 5.
3. **Math.pow(double a, double b)**
   * Returns the value of a raised to the power of b (i.e., a^b).
   * Example: Math.pow(2, 3) returns 8.0.
4. **Math.sqrt(double a)**
   * Returns the square root of the argument.
   * Example: Math.sqrt(25) returns 5.0.
5. **Math.min(double a, double b)**
   * Returns the smaller of two numbers.
   * Example: Math.min(5.0, 10.0) returns 5.0.
6. **Math.max(double a, double b)**
   * Returns the larger of two numbers.
   * Example: Math.max(5.0, 10.0) returns 10.0.
7. **Math.random()**
   * Returns a random double value between 0.0 and 1.0 (inclusive of 0.0 but not 1.0).
   * Example: Math.random() might return 0.54321.
8. **Math.PI**
   * A constant representing the value of π (approximately 3.141592653589793).
   * Example: Math.PI returns 3.14159....

**string concatenation**

In Java, **string concatenation** is the process of combining two or more strings into a single larger string. This is commonly done using the **+** or **+=** operators. Here's a breakdown of how string concatenation works in Java:

### String Concatenation Using +

The + operator can be used to concatenate strings, which means joining two or more strings together. When two String objects are combined using +, Java creates a new String object that holds the concatenated result.

* **Example**:

String greeting = "Hello, " + "world!";

**With Primitive Types**: When one of the operands is a string and the other is a primitive type (like int, double, or boolean), the primitive value is automatically converted into a string and concatenated. For example:

double result = 9.35;

String message = "Maximum is: " + result;

### Concatenation of Mixed Types

When concatenating a String with a non-string value (such as a number or boolean), Java automatically converts the non-string value into its string representation.

For example:

int count = 10;

String message = "There are " + count + " apples.";

**Booleans**: Similarly, when a boolean is concatenated with a string, it is converted to either "true" or "false":

boolean isActive = true;

String statusMessage = "Is the system active? " + isActive;

### String Concatenation Using +=

You can also use the += operator to append a string to an existing string:

String message = "Hello";

message += " world!";

This is equivalent to writing:

message = message + " world!";

It modifies the original string by appending the new string to it.

### Breaking Strings for Readability

If you have a very large string, you can break it into multiple smaller strings for better readability, and use concatenation to reassemble them. This is particularly useful in complex output statements or constructing long messages:

String longMessage = "This is a very long string " +

"that is broken into " +

"multiple parts for readability.";

### Implicit toString() Conversion

Whenever an object is concatenated with a String, the object's **toString()** method is implicitly called to get its string representation. This happens automatically, meaning you don't need to explicitly call toString() in your code.

### Method-Call Stack and Stack Frames

To understand how method calls work in Java, it’s helpful to understand the concept of the **method-call stack**, which is a key part of how the Java Virtual Machine (JVM) manages method execution and memory. This stack-based approach is used to keep track of the method calls, their local variables, and how to return control to the caller once a method finishes executing.

Here’s a breakdown of how the method-call stack works and the role of **stack frames**:

### What is a Stack?

A **stack** is a data structure that works on a **Last-In, First-Out (LIFO)** basis. This means that the last item pushed onto the stack is the first item to be popped off. To help visualize this, think of a stack of plates or dishes:

* When you **push** a dish onto the stack, it goes to the top.
* When you **pop** a dish off, you remove the dish from the top.

This concept of a stack is used by the method-call stack in programming, where methods are added and removed in the same way—each new method call gets pushed onto the stack, and when the method finishes executing, its data is popped off the stack.

### Method-Call Stack

The **method-call stack** is where the JVM stores information about the methods currently being executed. The stack keeps track of which method is calling which, and also manages local variables, including the parameters for each method call.

Each method that is invoked gets its own **stack frame**—a chunk of memory that holds:

1. **Local Variables**: These include any variables declared within the method (including method parameters).
2. **Return Address**: The memory address of the instruction to return to once the method has finished executing.
3. **Other Method Context Information**: This includes things like the method’s execution state.

### Stack Frames

A **stack frame** is the portion of the stack that contains all the necessary data for a specific method invocation. It holds:

* **Parameters**: Any values passed to the method.
* **Local Variables**: Any variables declared within the method.
* **Return Address**: The point in the code to return to after the method finishes.

When a method is called, a new stack frame is created and pushed onto the stack. When the method finishes executing, its stack frame is popped off the stack, and control is returned to the calling method.

Here’s how the method-call stack would work:

1. **Main method (main)** is called first:
   * A stack frame for main is pushed onto the stack.
   * It calls methodA.
2. **Method methodA** is called:
   * A stack frame for methodA is pushed onto the stack.
   * Inside methodA, a local variable x is created.
   * methodA then calls methodB.
3. **Method methodB** is called:
   * A stack frame for methodB is pushed onto the stack.
   * Inside methodB, a local variable y is created.
   * After methodB finishes execution, its stack frame is popped off the stack and control returns to methodA.
4. **Method methodA** finishes:
   * The stack frame for methodA is popped off the stack, and control returns to main.
5. **Main method (main)** finishes:
   * The stack frame for main is popped off the stack, and the program ends.

### Stack Overflow

A **stack overflow** occurs when there are too many methods calls and the method-call stack exceeds its memory limit. Since each method call uses a certain amount of memory to create a new stack frame, if you keep calling methods, eventually you will run out of space on the stack. This can happen if your program has a bug that causes infinite recursion, for example.

When a stack overflow occurs, an error is thrown, typically resulting in a **StackOverflowError**

### Java API Packages Overview

Java's **Application Programming Interface (API)** is a large set of predefined classes and interfaces grouped into **packages**. These packages provide a wide range of functionality, allowing developers to perform common tasks like file handling, networking, user interface design, and much more, without having to build everything from scratch.

In the following sections, we’ll explore some key Java API packages that are often used in Java programming.

### Common Java API Packages

1. **java.awt.event**: This package contains classes and interfaces related to **event handling** for Graphical User Interface (GUI) components in both the java.awt and javax.swing packages. It's commonly used for handling user interactions such as button clicks, mouse events, and keyboard input in GUI applications.

**Example:** Handling a button click in a GUI.

2. **java.awt.geom**: The **Java 2D Shapes Package** contains classes for creating and manipulating 2D geometric shapes such as lines, circles, and polygons. It is used for drawing graphics in Java applications, especially in custom GUI components or graphical applications.

* **Example:** Drawing circles, rectangles, and custom shapes.

**3. java.io:** The **Java Input/Output Package** provides classes for reading and writing data to files, streams, and other input/output resources. You use this package for handling file operations, such as reading from or writing to text files, binary files, or handling data streams.

* **Example:** Reading from a file using BufferedReader or writing to a file using FileWriter.

4. **java.lang**: The **Java Language Package** contains fundamental classes and interfaces essential for the core functionality of the language. It is automatically imported in every Java program. Contains basic functionality like string manipulation (String), math operations (Math), and system-level functionality (System, Thread).

* **Example:** Using System.out.println() to output data.

5. **java.net**: The **Java Networking Package** contains classes and interfaces that enable communication over computer networks, including the Internet. It’s used for writing network-based applications such as web clients and servers, and for accessing resources over HTTP or FTP.

* **Example:** Connecting to a server using Socket or URL.

6. **java.security**: The **Java Security Package** provides classes that implement security features, such as encryption, authentication, and data protection. It's used to secure data, manage cryptographic operations, and handle authentication processes in secure applications.

* **Example:** Creating a secure MessageDigest for hashing data.

**7. java.sql:** The **Java Database Connectivity (JDBC) Package** contains classes and interfaces for working with relational databases.It's used for connecting to databases, executing SQL queries, and processing result sets.

* **Example:** Connecting to a MySQL database using DriverManager and executing queries via Statement.

**8. java.util:** The **Java Utilities Package** contains a wide variety of utility classes, including collections (such as List, Set, and Map), date/time utilities, and more.It is commonly used for data manipulation, handling collections, performing sorting, and working with dates.

* **Example:** Using ArrayList to store a list of objects or HashMap for key-value pairs.

9. **java.util.concurrent**: The **Java Concurrency Package** provides utilities for handling concurrent programming, allowing multiple tasks to be performed in parallel. It is used to manage threads, synchronization, and other concurrency-related tasks.

* **Example:** Using ExecutorService to run tasks in parallel.

10. **javax.swing**

* **Description:** The **Java Swing GUI Components Package** is a part of Java's GUI library, providing lightweight, platform-independent components like buttons, text fields, and labels.
* **Usage:** It’s used to build desktop applications with graphical user interfaces.
* **Example:** Creating a JFrame window with buttons and labels.

11. **javax.swing.event**: This package deals with **event handling** for GUI components in the Swing library (part of javax.swing). It's used for listening to user interactions in Swing-based applications (e.g., clicking a button or selecting an item).

* **Example:** Responding to a button click with an ActionListener.

12. **javax.xml.ws**: The **JAX-WS Package** provides classes and interfaces for building and consuming **web services** in Java, particularly for SOAP-based web services. It’s used for web service development, such as creating and consuming web services that follow the SOAP protocol.

* **Example:** Creating a client that accesses a SOAP-based web service.

**Day 3**

### Method Overloading in Java

**Method overloading** occurs when multiple methods in the same class have the **same name** but differ in their **parameter lists**. These differences are typically in the number of parameters, the types of parameters, or the order of parameters. The return type of the methods can be the same or different, but it is not a factor in method overloading.

Java uses method overloading to allow methods to perform similar tasks on different types or numbers of inputs. When a method is called, the compiler determines which version of the overloaded method to invoke based on the arguments provided.

### Key Points of Method Overloading:

* **Same Method Name:** All overloaded methods share the same name.
* **Different Parameter List:** The parameter list must differ by:
  + Number of parameters
  + Type of parameters
  + Order of parameters
* **No Ambiguity:** Method overloading should not result in ambiguity. If two methods have the same name and the compiler cannot distinguish between them based on the arguments, it will cause a compile-time error.

**Random number generation**

**Using the Random class**

In Java, you can generate random numbers using the **Random** class, which is part of the **java.util** package. Random numbers are useful in a variety of applications like games, simulations, and security features.

The Random class in Java provides methods for generating random numbers. Below is a list of the most commonly used methods in the Random class, along with their descriptions and use cases:

1. **nextBoolean():** Returns a random boolean value: true or false. It Can be used to randomly decide between two options, e.g., simulating a coin toss.

**2. nextInt():** Returns a random integer (positive or negative) within the full range of the int type.ItWhen you need a random integer without bounds.

**3. nextInt(int bound)** : Returns a random integer between 0 (inclusive) and the specified bound (exclusive). It is Useful for generating random numbers within a specific range, like picking a random index in an array.

**4. nextLong()**: Returns a random long value.When a random long value is needed, such as for generating random timestamps or large numerical data.

**5. nextFloat()**: Returns a random float between 0.0 (inclusive) and 1.0 (exclusive).When you need random floating-point numbers, typically for simulation or random scoring.

**6. nextDouble()** : Returns a random double between 0.0 (inclusive) and 1.0 (exclusive).It is useful for generating random floating-point numbers with more precision than nextFloat().

**Using SecureRandom**

SecureRandom is a class in Java which is found in java.security package. It provides a cryptographically secure random number generator (RNG). Unlike the regular Random class, which generates pseudo-random numbers based on a seed value, SecureRandom produces numbers that are difficult to predict and more suitable for cryptographic applications, where unpredictability and security are crucial, making it suitable for tasks that require strong randomness, like generating encryption keys, tokens, passwords, or any other security-related applications.

**Introduction to Enums in Java**

In Java, an **enum** (short for "enumeration") is a special type used to define collections of constants. Enums are a powerful feature in Java that allow you to represent a fixed set of related constants in a more readable and maintainable way than traditional constants defined using public static final. For examples

* **Season**: Enum representing the four seasons: WINTER, SPRING, SUMMER, FALL.
* **Direction**: Enum representing the cardinal directions: NORTH, EAST, SOUTH, WEST.
* **Priority**: Enum representing the priority levels of tasks: LOW, MEDIUM, HIGH.
* **Day**: Enum representing the days of the week: SUNDAY, MONDAY, TUESDAY, WEDNESDAY, etc.
* **Month**: Enum representing the months of the year: JANUARY, FEBRUARY, MARCH, etc.
* **Status**: Enum representing the status of an order: PENDING, SHIPPED, DELIVERED, CANCELLED.
* **Color**: Enum representing basic colors: RED, GREEN, BLUE, YELLOW, BLACK, WHITE.
* **Size**: Enum representing clothing sizes: SMALL, MEDIUM, LARGE, XL, XXL.
* **PaymentMethod**: Enum representing various payment methods: CREDIT\_CARD, DEBIT\_CARD, PAYPAL, BANK\_TRANSFER, CASH.

Chapter 6

**Objectives:**

* Learn what primitive types and reference types are.
* Learn what arrays are.
* Use arrays to store data in and retrieve data from lists and tables of values.
* Declare arrays, initialize arrays and refer to individual elements of arrays.
* Iterate through arrays with the enhanced **for** statement.
* Pass arrays to methods.
* Declare and manipulate multidimensional arrays.
* Use variable-length argument lists.
* Read command-line arguments into a program.
* Search, sort and fill arrays with the methods of class Arrays, which contains methods for common array manipulations.
* Use class ArrayList to manipulate a dynamically resizable arraylike data structure.

**Day 4**

**Arrays**

An array is a group of variables (called elements or components) containing values that all have the same type. Arrays are objects, so they’re considered reference types. In Java, an array is a data structure that can hold a fixed number of values of a single type. Arrays are useful when you need to work with a collection of data elements, such as primitive type or objects. To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array. The position number of the element is called the element’s index or subscript.

**Types of Arrays**

There are two types of arrays in java

1. Single dimensional or One-dimensional Arrays
2. Multi-dimensional or two-dimensional arrays

**• One-dimensional arrays:** One-dimensional arrays in Java are linear lists of elements of the same type.

**Declaring and initializing Arrays**

### ****Declaring****

To declare a single-dimensional array, specify the type of elements it will hold, followed by square brackets:

int[] numbers; // Declaration of an integer array

### ****Creating****

After declaring an array, you need to create it with a specific size using the new keyword:

numbers = new int[5]; // Creates an array that can hold 5 integers

### ****Initializing****

You can initialize an array at the time of declaration:

int[] numbers = {1, 2, 3, 4, 5}; // Declaration and initialization

**Accessing and modifying Arrays**

### ****Accessing Elements:**** You can access elements in an array using their indices, which start at 0:

int firstNumber = numbers[0]; // Access the first element (1)

int secondNumber = numbers[1]; // Access the second element (2)

### ****Modifying or setting Elements:**** You can modify array elements by accessing them with their index:

numbers[2] = 10; // Sets the third element to 10

### ****Finding Array Length:**** The length of an array is obtained using the length attribute:

int length = numbers.length; // Gets the number of elements in the array

### ****Iterating through each element of the Arrays:**** You can use a for loop or an enhanced for statement or for each loop to iterate through each element of the array.

**Example**

Summing, multiplying, dividing, subtracting and finding modulus of an array

**Using Bar Charts to Display Array Data Graphically**

One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (\*). Suppose the grades on an exam were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. They include one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, array[0] indicates the number of grades in the range 0–9, array[7] the number of grades in the range 70–79 and array[10] the number of 100 grades.

Code

The GradeBook case studies in Chapter 7 contain code that calculates these grade

**Enhanced for statement**

In Java, the enhanced for statement, also known as the "for-each" loop. The enhanced for statement iterates through the elements of arrays or collections without using a counter, thus avoiding the possibility of “stepping outside” the array which eliminates the need for an explicit iterator or index variable, making the code more readable and less error-prone.

**Syntax**

for (Type item : collection) {

// Use item here

}

* Type: The type of elements in the collection (e.g., int, String, or a custom class).
* item: A temporary variable that takes on the value of each element in the collection during each iteration.
* collection: The array or collection you want to iterate over.

**Note**

The enhanced for statement can be used only to obtain array elements—it cannot be used to modify elements. If your program needs to modify elements, use the traditional counter-controlled for statement. The enhanced for statement can be used in place of the counter-controlled for statement whenever code looping through an array does not require access to the counter indicating the index of the current array element. For example, totaling the integers in an array requires access only to the element values—the index of each element is irrelevant. However, if a program must use a counter for some reason other than simply to loop through an array (e.g., to print an index number next to each array element value, use the counter-controlled for statement.

**Week 4**

**Day 1**

**Passing Arrays to Methods**

In Java, you can pass arrays to methods just like you pass other types of arguments. Passing arrays to methods allows you to manipulate or access array elements within the method. To pass an array argument to a method, specify the name of the array without any brackets.

For a method to receive an array reference through a method call, the method’s parameter list must specify an array parameter

### syntax

To pass an array to a method, you use the following syntax:

public void methodName(Type[] arrayParameter) {

// Method body

}

For example, if array hourlyTemperatures is declared as

double[] hourlyTemperatures = new double[24];

then the method call

modifyArray(hourlyTemperatures);

The method call passes array hourlyTemperature’s reference, so when the called method uses the array variable b, it refers to the same array object as hourlyTemperatures in the caller. When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference. However, when an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element’s value. Such primitive values are called scalars or scalar quantities. To pass an individual array element to a method, use the indexed name of the array element as an argument in the method call.

**Multidimensional arrays**

Multidimensional arrays with two dimensions are often used to represent tables of values with data arranged in rows and columns. To identify a particular table element, you specify two indices. By convention, the first identifies the element’s row and the second its column. Arrays that require two indices to identify each element are called two-dimensional arrays. (Multidimensional arrays can have more than two dimensions.)

**Declaration**

You declare a Multidimensional array using the following syntax:

Type[][] arrayName;

**Two-Dimensional Arrays with Rows of Different Lengths**

Java also supports jagged arrays, which are arrays of arrays where each sub-array can have a different length. This is useful when you need arrays with varying sizes.

int[][] b = {{1, 2}, {3, 4, 5}};

**Traverse a Multidimensional arrays.**

Multidimensional array is essentially an array of arrays. To traverse it, you use nested for loops

**Arrays Class**

The Arrays class in Java provides static methods for manipulating arrays, such as sorting, searching, and comparing arrays. It simplifies common tasks that involve arrays. This class is **final** and contains only static methods. It does **not** represent an array itself, but rather provides utility methods for working with arrays. It is found inside java.util package. Some of the most commonly used methods in the Arrays class:

**1. Sorting Arrays**

* **sort Method**: Sorts an array into ascending order.

int[] numbers = {5, 3, 8, 1, 2};

Arrays.sort(numbers); // numbers is now {1, 2, 3, 5, 8}

You can also sort a specific range of an array:

Arrays.sort(numbers, 1, 4); // Sorts elements from index 1 to index 3

**2. Searching Arrays**

* **binarySearch Method**: Searches for a specified value using binary search (requires the array to be sorted).

int index = Arrays.binarySearch(numbers, 5); // Returns the index of the value 5

**3. Equality and Comparison**

* **equals Method**: Checks if two arrays are equal (same size and elements).

int[] arr1 = {1, 2, 3};

int[] arr2 = {1, 2, 3};

boolean isEqual = Arrays.equals(arr1, arr2); // Returns true

* **compare Method**: Compares two arrays lexicographically.

int result = Arrays.compare(arr1, arr2); // Returns 0 if they are equal

**4. Filling Arrays**

* **fill Method**: Fills all elements of an array with a specified value
* int[] numbers = new int[5];

Arrays.fill(numbers, 42); // Fills all elements with 42

You can also fill a specific range:

Arrays.fill(numbers, 1, 4, 99); // Fills elements from index 1 to 3 with 99

**5. Copying Arrays**

* **copyOf Method**: Copies an array, possibly resizing it.

int[] original = {1, 2, 3};

int[] copy = Arrays.copyOf(original, 5); // New array with length 5, last two elements are 0

**6. Converting Arrays to Strings**

* **toString Method**: Returns a string representation of an array.

String[] fruits = {"apple", "banana", "cherry"};

String str = Arrays.toString(fruits); // "[apple, banana, cherry]"

* **deepToString Method**: For arrays with nested arrays, returns a string representation of the entire array structure.

int[][] matrix = {{1, 2}, {3, 4}};

String str = Arrays.deepToString(matrix); // "[[1, 2], [3, 4]]"

**Reference type**

In Java, a **reference type** is a type that refers to or points to an object rather than holding the actual data directly. Think of it like having a list of addresses (reference types) instead of the actual houses (objects).

For example, if you have a String in Java, it's a reference type. The variable doesn’t store the actual string of characters, but rather it stores the address where the string is located in memory.

**Day 2**

**variable-length argument lists**

Variable-length argument lists (varargs) is used to create methods that accept a variable number of arguments of the same type. This feature is useful when you want to provide flexibility in the number of arguments a method can handle, without needing to overload the method for different numbers of parameters.

### Syntax for Varargs

The syntax for declaring a method with varargs is to use an ellipsis (...) after the type of the parameter. This tells the Java compiler that the method can accept zero or more arguments of that type.

public void methodName(Type... args) {

// Method body

}

**Key Points**

1. **Varargs Must Be the Last Parameter:**
   * If you use varargs in a method, it must be the last parameter in the parameter list. You can only have one varargs parameter per method.
2. **Varargs are Treated as Arrays:**
   * Within the method, the varargs parameter is treated as an array. You can iterate over it, access elements by index, and use it like any other array.

**Using Command-Line Arguments**

In Java, command-line arguments allow you to pass information to your program at runtime. These arguments are provided as an array of String objects to the main method of your program. This is useful for making your application flexible and interactive, as you can control its behavior through command-line inputs.

**Syntax**

When you run a Java program, command-line arguments are passed to the main method. The main method in Java is defined as follows:

public static void main(String[] args) {

// Code here

}

* args is an array of String objects that holds the command-line arguments.

**Introduction to Collections and Class ArrayList<T>, Generic Types, and Wrapper Classes in Java**

**What is an ArrayList?**

### An ArrayList is a type of collection that is used to store a list of items (like an array). However, unlike an array, an ArrayList is dynamic, which means it can grow and shrink in size as needed. You don't have to specify its size when you create it. The ArrayList class is found in java.util package.

Syntax:

ArrayList<String> list = new ArrayList<>(); // Stores only Strings

**What are Generics in Java?**

* **Generics** allow you to write **flexible and reusable code** that works with different types while still ensuring type safety.
* A **generic type** is a placeholder for a data type. When you define a generic class or method, you can use any object type in place of the placeholder.

Example of a **generic ArrayList**:

ArrayList<Boolean> intList = new ArrayList<>(); // Stores only Integers

intList.add(10);

Here, Integer is the **specific type** used to replace the **generic type** placeholder (T).

**Why Use Generics?**

* **Type Safety**: Ensures the right type of objects are added to a collection. It prevents bugs like trying to add a wrong type of object (e.g., adding a String to a list of Integers).
* **Reusability**: You can create one class or method that works with multiple types, without repeating code for each specific type.

**What are Wrapper Classes?**

* Java’s **primitive types** (like int, char, double, etc.) cannot be used directly with generics, because generics only work with **objects** (not primitive types).
* **Wrapper classes** are special classes in Java that "wrap" primitive types into objects.

### ****Primitive Types and Their Wrapper Classes in Java:****

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper Class** |
| Boolean | Boolean |
| Byte | Byte |
| Short | Short |
| Int | Integer |
| Long | Long |
| Float | Float |
| Double | Double |
| Char | Character |

* **Boxing**: Converting a primitive to a wrapper class (e.g., int to Integer).
* **Unboxing**: Converting a wrapper class back to a primitive (e.g., Integer to int).

Key Points

1. **Array vs. ArrayList**:
   * **Array**: Fixed in size. Once you declare an array, the size can’t change.
   * **ArrayList**: Flexible. You can add and remove items, and the list will adjust automatically.
2. **Key Features of ArrayList**:
   * It allows **random access** to elements (just like an array).
   * You can **add**, **remove**, **get**, and **set** elements easily.
   * It grows or shrinks as you add or remove elements.

**Method and their Description**

Here’s a summary of some common methods available in the ArrayList class:

**Commonly Used Methods**

1. **Adding Elements:**
   * add(E e): Appends the specified element to the end of the list.
   * add(int index, E element): Inserts the specified element at the specified position in the list.
2. **Accessing Elements:**
   * get(int index): Returns the element at the specified position in the list.
   * set(int index, E element): Replaces the element at the specified position in the list with the specified element.
3. **Removing Elements:**
   * remove(Object o): Removes the first occurrence of the specified element from the list (if present).
   * remove(int index): Removes the element at the specified position in the list.
4. **Size and Capacity:**
   * size(): Returns the number of elements in the list.
   * isEmpty(): Checks if the list is empty.
5. **Clearing the List:**
   * clear(): Removes all of the elements from the list.
6. **Index and Contains:**
   * indexOf(Object o): Returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
   * contains(Object o): Checks if the list contains the specified element.
7. **Sublist and View:**
   * subList(int fromIndex, int toIndex): Returns a view of the portion of the list between the specified fromIndex, inclusive, and toIndex, exclusive.
8. **List Iteration:**
   * iterator(): Returns an iterator over the elements in the list.
   * forEach(Consumer<? super E> action): Performs the given action for each element of the list.
9. **Conversion and Copying:**
   * toArray(): Returns an array containing all of the elements in the list.

# **Date and Time**

In Java, **date and time** refer to representations of specific moments or spans in time. Java has long provided classes for working with dates and times, but these were often found to be cumbersome, error-prone, and not thread-safe. Starting with **Java 8,** the **Date and Time API** was introduced in the java.time package to address these issues. The new API is modern, clear, and much more flexible than previous implementations. It also provides greater support for different time zones, formatting, and parsing.

**1. LocalDate (Class)**

LocalDate is a class from the java.time package, which is part of the Java 8 Date and Time API. It represents a date without time or timezone information (i.e., it includes only year, month, and day).

### Key Features:

* **Immutability**: LocalDate objects are immutable, meaning once a LocalDate object is created, its value cannot be changed.
* **No time or timezone**: It only contains the date (year, month, day) without any time or timezone information.
* **Useful for date-based calculations**: It is used when you need to work with dates in a specific calendar system (usually the ISO-8601 calendar).

### Common Methods of LocalDate:

1. **now()**
   * public static LocalDate now()
   * This method returns the current date from the system clock in the default time zone.
2. **of()**
   * public static LocalDate of(int year, int month, int dayOfMonth)
   * This method creates a LocalDate instance from the given year, month, and day.
3. **parse()**
   * public static LocalDate parse(CharSequence text)
   * This method parses a string representing a date in ISO-8601 format (YYYY-MM-DD) and returns a LocalDate.
4. **getYear()**
   * public int getYear()
   * Returns the year of the LocalDate.
5. **getMonth()**
   * public Month getMonth()
   * Returns the month as a Month enum (e.g., FEBRUARY).
6. **getDayOfMonth()**
   * public int getDayOfMonth()
   * Returns the day of the month.
7. **plusDays()**
   * public LocalDate plusDays(long days)
   * This method returns a new LocalDate by adding the specified number of days to the current date.
8. **minusDays()**
   * public LocalDate minusDays(long days)
   * This method returns a new LocalDate by subtracting the specified number of days from the current date.
9. **isLeapYear()**
   * public boolean isLeapYear()
   * This method returns true if the year of the LocalDate is a leap year.
10. **equals()**
    * public boolean equals(Object obj)
    * Compares the LocalDate object to another object for equality.

**2**. **LocalTime (Class)**

LocalTime is a class from the java.time package, introduced in Java 8, that represents a time without a date or timezone. It holds time information (hours, minutes, seconds, and nanoseconds) but does not contain date or timezone data.

### Key Features:

* **Immutability**: LocalTime objects are immutable, meaning once a LocalTime object is created, its value cannot be changed.
* **No date or timezone**: It only contains the time portion (hour, minute, second, nanosecond) without a date or timezone information.
* **ISO-8601 standard**: LocalTime uses the ISO-8601 standard for time representation (hh:mm:ss.nnnnnnnnn).

### Common Methods of LocalTime:

1. **now()**
   * public static LocalTime now()
   * This method returns the current time from the system clock in the default time zone.

**Example**:

LocalTime now = LocalTime.now();

System.out.println(now); // Prints current time, e.g., 14:35:50.345

1. **of()**
   * public static LocalTime of(int hour, int minute)
   * public static LocalTime of(int hour, int minute, int second)
   * public static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
   * This method creates a LocalTime instance using the provided values for hour, minute, second, and nanosecond.

**Example**:

LocalTime time = LocalTime.of(14, 30);

System.out.println(time); // Prints: 14:30

1. **parse()**
   * public static LocalTime parse(CharSequence text)
   * This method parses a string representing a time in ISO-8601 format (HH:mm:ss) or a custom format and returns a LocalTime.

**Example**:

LocalTime parsedTime = LocalTime.parse("14:30:45");

System.out.println(parsedTime); // Prints: 14:30:45

1. **getHour()**
   * public int getHour()
   * Returns the hour value of the LocalTime object.

**Example**:

int hour = LocalTime.now().getHour();

System.out.println(hour); // Prints the current hour (0-23)

1. **getMinute()**
   * public int getMinute()
   * Returns the minute value of the LocalTime object.

**Example**:

int minute = LocalTime.now().getMinute();

System.out.println(minute); // Prints the current minute (0-59)

1. **getSecond()**
   * public int getSecond()
   * Returns the second value of the LocalTime object.

**Example**:

int second = LocalTime.now().getSecond();

System.out.println(second); // Prints the current second (0-59)

1. **getNano()**
   * public int getNano()
   * Returns the nanosecond value of the LocalTime object.

**Example**:

int nano = LocalTime.now().getNano();

System.out.println(nano); // Prints the current nanoseconds (0-999999999)

1. **plusHours()**
   * public LocalTime plusHours(long hours)
   * Returns a new LocalTime after adding the specified number of hours to the current time.

**Example**:

LocalTime newTime = LocalTime.now().plusHours(2);

System.out.println(newTime); // Prints the time after adding 2 hours

1. **plusMinutes()**
   * public LocalTime plusMinutes(long minutes)
   * Returns a new LocalTime after adding the specified number of minutes to the current time.

**Example**:

LocalTime newTime = LocalTime.now().plusMinutes(30);

System.out.println(newTime); // Prints the time after adding 30 minutes

1. **minusMinutes()**
   * public LocalTime minusMinutes(long minutes)
   * Returns a new LocalTime after subtracting the specified number of minutes from the current time.

**Example**:

LocalTime newTime = LocalTime.now().minusMinutes(15);

System.out.println(newTime); // Prints the time after subtracting 15 minutes

1. **isBefore()**
   * public boolean isBefore(LocalTime other)
   * Compares if the current LocalTime is before another LocalTime.

**Example**:

boolean isBefore = LocalTime.now().isBefore(LocalTime.of(12, 0));

System.out.println(isBefore); // Prints true or false

1. **isAfter()**
   * public boolean isAfter(LocalTime other)
   * Compares if the current LocalTime is after another LocalTime.

**Example**:

boolean isAfter = LocalTime.now().isAfter(LocalTime.of(12, 0));

System.out.println(isAfter); // Prints true or false

1. **equals()**
   * public boolean equals(Object obj)
   * Compares the LocalTime object to another object for equality.

**Example**:

LocalTime time1 = LocalTime.of(14, 30);

LocalTime time2 = LocalTime.of(14, 30);

boolean isEqual = time1.equals(time2);

System.out.println(isEqual); // Prints: true

**3.** L**ocalDateTime (Class)**

LocalDateTime is a class from the java.time package that represents a date-time without a timezone. It combines both LocalDate (which holds the date) and LocalTime (which holds the time) into a single object, allowing you to work with both date and time in a specific format (year, month, day, hour, minute, second, and nanoseconds) without any timezone information.

**Key Features:**

* **Immutability**: LocalDateTime objects are immutable, meaning once created, their values cannot be changed.
* **No timezone**: Unlike ZonedDateTime or OffsetDateTime, LocalDateTime doesn't store any timezone information. It only represents the local date and time in the system's default time zone.
* **ISO-8601 standard**: It follows the ISO-8601 standard for date-time representation (YYYY-MM-DDTHH:MM:SS.nnnnnnnnn).

**Common Methods of LocalDateTime:**

1. **now()**
   * public static LocalDateTime now()
   * This method returns the current date-time from the system clock in the default time zone.

**Example**:

LocalDateTime now = LocalDateTime.now();

System.out.println(now); // Prints the current date-time

1. **of()**
   * public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
   * public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)
   * public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond)
   * This method creates a LocalDateTime instance with the specified values for year, month, day, hour, minute, second, and nanosecond.

**Example**:

LocalDateTime dateTime = LocalDateTime.of(2025, 2, 6, 14, 30);

System.out.println(dateTime); // Prints: 2025-02-06T14:30

1. **parse()**
   * public static LocalDateTime parse(CharSequence text)
   * This method parses a string representing a date-time in ISO-8601 format and returns a LocalDateTime.

**Example**:

LocalDateTime parsedDateTime = LocalDateTime.parse("2025-02-06T14:30:00");

System.out.println(parsedDateTime); // Prints: 2025-02-06T14:30

1. **getYear()**
   * public int getYear()
   * Returns the year from the LocalDateTime object.

**Example**:

int year = LocalDateTime.now().getYear();

System.out.println(year); // Prints the current year (e.g., 2025)

1. **getMonth()**
   * public Month getMonth()
   * Returns the month as a Month enum.

**Example**:

Month month = LocalDateTime.now().getMonth();

System.out.println(month); // Prints the current month (e.g., FEBRUARY)

1. **getDayOfMonth()**
   * public int getDayOfMonth()
   * Returns the day of the month.

**Example**:

int dayOfMonth = LocalDateTime.now().getDayOfMonth();

System.out.println(dayOfMonth); // Prints the current day of the month

1. **getHour()**
   * public int getHour()
   * Returns the hour (0-23) from the LocalDateTime object.

**Example**:

int hour = LocalDateTime.now().getHour();

System.out.println(hour); // Prints the current hour (0-23)

1. **plusDays()**
   * public LocalDateTime plusDays(long days)
   * This method adds the specified number of days to the LocalDateTime.

**Example**:

LocalDateTime newDateTime = LocalDateTime.now().plusDays(5);

System.out.println(newDateTime); // Prints the date-time 5 days from now

1. **plusHours()**
   * public LocalDateTime plusHours(long hours)
   * This method adds the specified number of hours to the LocalDateTime.

**Example**:

LocalDateTime newDateTime = LocalDateTime.now().plusHours(2);

System.out.println(newDateTime); // Prints the date-time 2 hours from now

1. **minusMinutes()**
   * public LocalDateTime minusMinutes(long minutes)
   * This method subtracts the specified number of minutes from the LocalDateTime.

**Example**:

LocalDateTime newDateTime = LocalDateTime.now().minusMinutes(15);

System.out.println(newDateTime); // Prints the date-time 15 minutes ago

1. **isBefore()**
   * public boolean isBefore(LocalDateTime other)
   * Compares if the current LocalDateTime is before another LocalDateTime.

**Example**:

boolean isBefore = LocalDateTime.now().isBefore(LocalDateTime.of(2025, 2, 6, 12, 0));

System.out.println(isBefore); // Prints true or false

1. **isAfter()**
   * public boolean isAfter(LocalDateTime other)
   * Compares if the current LocalDateTime is after another LocalDateTime.

**Example**:

boolean isAfter = LocalDateTime.now().isAfter(LocalDateTime.of(2025, 2, 6, 12, 0));

System.out.println(isAfter); // Prints true or false

1. **equals()**
   * public boolean equals(Object obj)
   * Compares the LocalDateTime object to another object for equality.

**Example**:

**4. DateTimeFormatter (Class)**

In Java, DateTimeFormatter is a class from the java.time.format package, introduced in Java 8, that is used for formatting and parsing date-time objects. It allows you to convert LocalDate, LocalTime, LocalDateTime, ZonedDateTime, and other java.time objects to and from their string representations based on a pattern or predefined styles.

**Key Features of DateTimeFormatter:**

* **Formatting**: It can format date-time objects into readable string representations using custom patterns.
* **Parsing**: It can parse strings into date-time objects according to specific patterns.
* **Pattern-based formatting**: You can define patterns using symbols like yyyy, MM, dd, HH, mm, ss, etc., to format or parse date and time in various ways.
* **Predefined formats**: It provides a few predefined formats like ISO\_LOCAL\_DATE, ISO\_LOCAL\_TIME, ISO\_LOCAL\_DATE\_TIME, and more.

**Common Methods of DateTimeFormatter:**

1. **ofPattern(String pattern)**
   * **Definition**: This method creates a DateTimeFormatter instance with a custom pattern string.
   * **Usage**: The pattern defines the format to which the date-time object should be formatted or from which it should be parsed. For example, yyyy-MM-dd for a date or HH:mm:ss for a time.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

String formattedDate = LocalDateTime.now().format(formatter);

System.out.println(formattedDate); // Example output: 2025-02-06 14:30:45

1. **ofPattern(String pattern, Locale locale)**
   * **Definition**: Similar to ofPattern(String pattern), but it allows specifying the locale to support regional formatting styles.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMM yyyy", Locale.FRANCE);

String formattedDate = LocalDate.now().format(formatter);

System.out.println(formattedDate); // Example output: 06 févr. 2025

1. **ISO\_DATE**
   * **Definition**: A predefined DateTimeFormatter for the ISO-8601 date format yyyy-MM-dd.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ISO\_DATE;

String formattedDate = LocalDate.now().format(formatter);

System.out.println(formattedDate); // Example output: 2025-02-06

1. **ISO\_LOCAL\_DATE**
   * **Definition**: A predefined DateTimeFormatter for the ISO-8601 date format without time (yyyy-MM-dd).

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ISO\_LOCAL\_DATE;

String formattedDate = LocalDate.now().format(formatter);

System.out.println(formattedDate); // Example output: 2025-02-06

1. **parse(CharSequence text)**
   * **Definition**: Parses a string representation of a date-time into a date-time object according to the format defined by the DateTimeFormatter.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

LocalDateTime dateTime = LocalDateTime.parse("2025-02-06 14:30:45", formatter);

System.out.println(dateTime); // Example output: 2025-02-06T14:30:45

1. **parse(CharSequence text, TemporalQuery<T> query)**
   * **Definition**: This method allows parsing a string and returning a result of the specified type using a TemporalQuery. It is often used when you want to retrieve specific information, like the ChronoField or a custom object from a parsed date-time.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

LocalDate date = LocalDate.parse("2025-02-06", formatter);

System.out.println(date); // Example output: 2025-02-06

1. **format(TemporalAccessor temporal)**
   * **Definition**: Formats a given TemporalAccessor (like LocalDate, LocalTime, or LocalDateTime) according to the format defined by the DateTimeFormatter.

**Example**:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy");

String formattedDate = LocalDate.now().format(formatter);

System.out.println(formattedDate); // Example output: 02/06/2025

**DateTimeFormatter Symbols:**

When creating a custom pattern for formatting or parsing date-time values, DateTimeFormatter uses several symbols to represent date and time components. Some common symbols are:

* y: Year (e.g., yyyy for 4 digits, yy for 2 digits)
* M: Month of year (e.g., MM for 2 digits, MMM for abbreviated month name)
* d: Day of month (e.g., dd for 2 digits)
* H: Hour of day (0-23, e.g., HH for 2 digits)
* m: Minute of hour (e.g., mm for 2 digits)
* s: Second of minute (e.g., ss for 2 digits)
* S: Millisecond (e.g., SSS for 3 digits)
* a: AM/PM marker (e.g., a for AM/PM)
* E: Day of week (e.g., E for abbreviated name, EEEE for full name)

**Chapter 7**

**Introduction to Classes and Objects**

**Day 3**

**Object-Oriented Programming (OOP)** is a programming style that organizes code around real-world objects and scenarios. It allows you to create "objects" that represent real-world things, each with its own properties and behaviors. These objects can interact with each other, just like how things in the real world interact.

OOP uses concepts like

* **classes,**
* **encapsulation,**
* **inheritance,**
* **polymorphism,**
* **and abstraction**

to structure and simplify code, making it easier to understand, maintain, and reuse.

Object-Oriented Programming (OOP) in Java centered around the concept of "objects," which can contain both data (attributes) and methods (functions or behaviors). In a layman term Object oriented programming (OOP) means the ability for a program to mimic the real world.

**Classes and objects**

**What is a Class in Java?**

Everything in Java is associated with classes and objects, along with its attributes and methods.

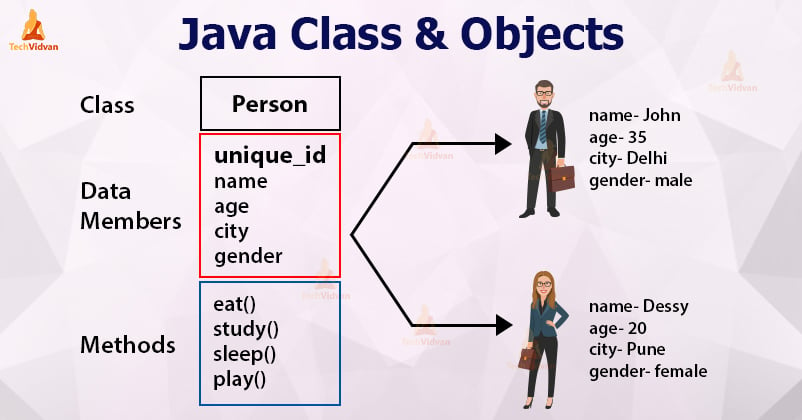
Think of a class in Java as a blueprint or a recipe. Just like a recipe tells you how to make a cake (what ingredients you need and what steps to follow), a class tells the computer how to create something and what it can do.

1. **Blueprint for Objects**:
   * A class defines what an object (or instance) of that class will look like and what it can do. For example, if you have a class called Dog, it will describe what makes up a dog (like its name, breed, age) and what a dog can do (like bark, eat).
2. **Attributes**:
   * Inside the class, you can specify attributes (also called fields or properties). These are the characteristics of the object. For a Dog class, attributes might include name, age, and breed.
3. **Methods**:
   * Methods are the actions or behaviors that the object can perform. In the Dog class, methods might include bark() or eat().

**Note:**

In programming, when we talk about creating objects, it's not like we're physically seeing a dog or a cake being made in front of us. Instead, the **object** exists in the computer's memory, and the "creation" of an object happens when the program runs.

So, when we say that an object is created, we mean that the program is allocating space in memory to store the data for that object. The object will have its **attributes** (like breed and age for a dog), and it can **perform actions** (like barking), but you won’t physically see it like you would a real object.



**What is an object in Java?**

An object represents an entity/object in the real world. An object has state, behavior and identity.

Let's break down the concepts of state, behavior, and identity using the examples mentioned below:

**1. Student Object:**

* **State**: Includes attributes such as student ID, name, age, courses enrolled, etc. These are the characteristics that describe a particular student.
* **Behavior**: Methods that a Student object can perform, such as enrollInCourse(), dropCourse(), study(), takeExam(), etc. These methods define what actions a student can take.
* **Identity**: Each student has a unique identity, typically represented by their student ID or a combination of unique attributes like name and ID.

**2. Circle Object:**

* **State**: Includes attributes like radius, center coordinates, and possibly color. These attributes describe the shape and properties of the circle.
* **Behavior**: Methods such as calculateArea(), calculateCircumference(), resize(), etc. These methods define what actions can be performed with the circle.
* **Identity**: Circles are identified by their unique properties such as radius and center coordinates.

**4. Bank Account Object:**

* **State**: Attributes include account number, balance, owner name, account type, etc., which describe the account and its financial status.
* **Behavior**: Methods such as deposit(), withdraw(), checkBalance(), transferFunds(), etc., which define what actions can be performed on the bank account.
* **Identity**: Each bank account is uniquely identified by its account number and possibly other identifying information like BVN.

**Declare a class and use it to create an object.**

* **Class Declaration (BankAccount)**:
  + Contains private fields (name, accountNumber, balance) to store account data.
  + Includes a constructor to initialize these fields.
  + Provides public methods (deposit, withdraw, getName, setName, getBalance, setBalance, getAccountNumber, setAccountNumber) to manipulate and access the data.
* **Object Creation (BankAccountTest):**
  + An instance of BankAccount is created with new BankAccount(“Jack”,"123456789", 1000.00).
  + The object myAccount is used to call methods and interact with the bank account.

**Instance variable**

An **instance variable** is a variable that is declared within a class but outside of any method or constructor. It represents a property or characteristic of an object created from that class. Each object (or instance) of the class has its own copy of the instance variables, and their values can differ from one object to another.

Here are some key points about instance variables:

1. **Defined at the class level**: They are declared directly inside the class, but outside of methods, constructors, or blocks.
2. **Unique to each object**: Every instance of a class has its own set of instance variables.
3. **Accessed by object**: They can be accessed using the dot (.) operator via an instance of the class.
4. **Default values**: If not explicitly initialized, instance variables are automatically given default values (e.g., null for objects, 0 for numeric types, false for booleans).

**Encapsulation**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), and it's essential for maintaining the integrity of data and ensuring that the internal workings of an object are hidden from the outside world.

**Encapsulation means** making sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as **private**
* provide public **get** and **set** methods to access and update the value of a private variable

**Getter and Setter Methods**

The "get" and "set" methods are commonly used to provide controlled access to an object's fields. They are part of the encapsulation principle of object-oriented programming, which helps to protect the internal state of an object and allows for controlled modification.

**Getter Method**: Retrieves the value of a private field. It usually has the prefix **get** followed by the field name with the first letter capitalized.

**Setter Method:** Sets or updates the value of a private field. It typically has the prefix **set** followed by the field name with the first letter capitalized.

**Constructors**

A constructor in Java is **a special method** that is automatically called when you create an instance (object) of a class. Its main job is to initialize the new object, which often involves setting initial values for its attributes (fields).

**Key Points About Constructors**

1. **Same Name as Class:** The constructor has the same name as the class in which it is defined.
2. **No Return Type:** Unlike regular methods, constructors do not have a return type, not even void.
3. **Called Automatically:** It is called automatically when an object is created using the new keyword.

**Types of Constructors**

1. **Default Constructor:** If you don't define any constructors in your class, Java provides a default constructor that initializes all numeric fields to 0, boolean fields to false, and object references to null.
2. **Parameterized Constructor:** You can define a constructor that takes arguments. This allows you to initialize the object with specific values right when it is created.

**Chapter 8**

**Day 4**

**Constructor overloading**

Constructor overloading is a concept in object-oriented programming (OOP) where a class has more than one constructor with different parameter lists. It allows an object to be initialized in multiple ways. This is particularly useful when you need to create instances of a class with different initial values or configurations.

**How it works:**

1. **Different Parameter Lists**: Each constructor must have a unique parameter list. This means that constructors can differ by the number or type of their parameters.
2. **No Return Type**: Constructors do not have a return type, not even void.
3. **Initialization**: Constructor overloading helps in initializing objects with different sets of values or providing different ways to initialize the state of an object.

**Association**

Association in Java is a connection or relation between two separate classes that are set up through their objects. Association relationship indicates how objects know each other and how they are using each other's functionality.

This relationship between classes can be

1. One-to-one: This is when one object is associated with exactly one object of another class.

Example: A person has one passport.

class Person {

Passport passport;

}

class Passport {

String passportNumber;

}

1. One-to-many: This is when one object is associated with multiple objects of another class.

Example: A teacher can have many students.

class Teacher {

List<Student> students;

}

class Student {

String name;

}

1. Many-to-one: This is when many objects of one class are associated with a single object of another class.

Example: Many employees work for one company.

class Employee {

Company company;

}

class Company {

String companyName;

}

1. Many-to-many: This is when many objects of one class are associated with many objects of another class.

Example: A student can enroll in many courses, and a course can have many students.

class Student {

List<Course> courses;

}

class Course {

List<Student> students;

}

**There are two types of association**

1. IS-A Association: The IS-A Association is also referred to as Inheritance.
2. HAS-A Association classified into two parts,
   1. Aggregation
   2. Composition

They are used to model relationships between classes, but they differ in terms of the strength and nature of these relationships.

**Aggregation**

**Definition**: Aggregation represents a "has-a" relationship where the child (or part) can exist independently of the parent (or whole). It indicates a weaker relationship between the objects.

**Key points**

**Lifetime**: In aggregation, the lifecycle of the child object is independent of the parent object. If the parent object is destroyed, the child object can still exist.

**Implementation**: Aggregation is implemented using a reference to another object. The child object can be shared among multiple parents.

### Examples of Aggregation

1. **Library and Books**:

A **Library** contains **Books**. Books can exist independently of the library, e.g., you can own a book without it being in a library.

1. **Team and Players**:

A **Team** consists of **Players**. Players can belong to multiple teams over time and exist independently of any particular team.

1. **School and Students**:

A **School** has **Students**. Students can transfer to different schools or leave school but are still individuals on their own.

1. **Company and Employees**:

A **Company** employs **Employees**. Employees can work for different companies throughout their careers.

1. **Department and Employees**:

A **Department** within a company has **Employees**. Employees can move to different departments or leave the company but the department itself exists independently.

**Composition**

**Definition**: Composition represents a stronger "has-a" relationship where the child object is tightly bound to the parent. It indicates that the child object's lifecycle is dependent on the parent object.

**Key Points**

**Lifetime**: In composition, if the parent object is destroyed, the child object is also destroyed. The child object cannot exist without the parent.

**Implementation**: Composition is implemented by creating the child object within the parent object, typically in the constructor of the parent. The child is not shared and is usually created and managed exclusively by the parent.

### Examples of Composition

1. **Book and Chapters:**

A Book is composed of Chapters. Chapters are parts of the book and do not exist outside the context of the book.

1. **Car and Engine:**

A Car has an Engine. The engine is a fundamental component of the car; if the car is destroyed, the engine is too.

1. **Tree and Leaves:**

A Tree has Leaves. Leaves are part of the tree and cannot exist independently in their typical role.

1. **Building and Walls:**

A Building is constructed with Walls. The walls are integral to the building’s structure and are not typically independent of it.

1. **Project and Tasks:**

A Project is composed of Tasks. Tasks are integral parts of the project and do not exist independently outside the project context.

**Week 5**

**Day 1**

**Static class members**

The concept of static keyword is used to define class-level members and methods that can be accessed without creating an instance of the class. Here's a breakdown of how static variables, static methods, and static initialization blocks work:

**1. Static Variables**

Static variables are also known as class variables. They are shared among all instances of a class. A single copy of the static variable exists, regardless of how many instances of the class are created.

* **Declaration**: static <dataType> <variableName>;

**2. Static Methods**

Static methods belong to the class rather than any specific instance. They can be called directly using the class name and can only access static variables and static methods.

* **Declaration**: static <returnType> <methodName>(<parameters>) { ... }

**3. Static Initialization Block**

A static initialization block is a special block of code that runs only once when the class is first loaded into memory. It's typically used to initialize static variables or to perform setup operations that need to be done before any static methods or fields are accessed.

**Declaration**: static { // Initialization code }

**Final instance variables**

The**final keyword is**used for classes, fields and methods, which makes them non-changeable i.e**., we cannot inherit, reassign or override them which** indicates that the value of the variable is constant after it has been assigned.

**Characteristics of Final Instance Variables**

1. **Initialization**:
   * A final instance variable must be initialized before it can be used. It can be initialized either at the point of declaration or in the constructor of the class.
2. **Assignment**:
   * Once a final instance variable has been assigned a value, it cannot be changed. Attempting to reassign a value to a final variable will result in a compilation error.
3. **Instance-specific**:
   * Unlike static variables, final instance variables are specific to each instance of the class. Each object has its own copy of the final instance variable.
4. **Usage**:
   * Final instance variables are typically used to represent constants or immutable data associated with an object.

**Inner classes**

Inner classes (also known as nested classes) are classes defined within another class. They are best for the purpose of logically grouping classes that are used in one-place. For example, if you want to create class which is used by ONLY enclosing class, then it doesn't make sense to create a separate file for that. Instead, you can add it as "inner class".

Compelling reasons for using nested classes include the following:

* It is a way of logically grouping classes that are only used in one place.
* It increases encapsulation.
* It can lead to more readable and maintainable code.

Java provides several types of inner classes, each serving different purposes. Inner classes can access the members of their outer class, which can be useful in certain design patterns and scenarios. The following are different types of inner classes in Java:

### 1. ****Regular Inner Class ( Also known as non-static inner class)****

A member inner class is defined within the body of another class but outside of any methods, constructors, or blocks. It has access to both static and instance variables and methods of the enclosing class.

### 2. ****Static Inner Class****

A static nested class is defined with the static keyword. Unlike non-static inner classes, it does not have access to the instance variables and methods of the enclosing class. It can only access the static members of the enclosing class.

### 3. Method ****Local Inner Class****

Local inner class is a class defined within a method or a block of code. It is local to that method or block, meaning it can only be used within that method or block where it is defined. Local inner classes have access to the final (or effectively final) variables of the enclosing method.

**Anonymous Inner Class**

### Key Points:

* **Access to Enclosing Class Members**:
  + **Member Inner Classes**: Can access all members (both static and instance) of the outer class.
  + **Static Nested Classes**: Can only access static members of the outer class.
  + **Local Inner Classes**: Can access final or effectively final local variables of the enclosing method.
* **Use Cases**:
  + **Member Inner Classes**: When you need a class that’s closely related to an instance of the outer class.
  + **Static Nested Classes**: When you need a helper class that doesn’t need to interact with instance members of the outer class.
  + **Local Inner Classes**: For temporary use within a method, usually when you need a class with a limited scope.

Using inner classes effectively can help organize code, improve encapsulation, and manage complexity by logically grouping related classes.

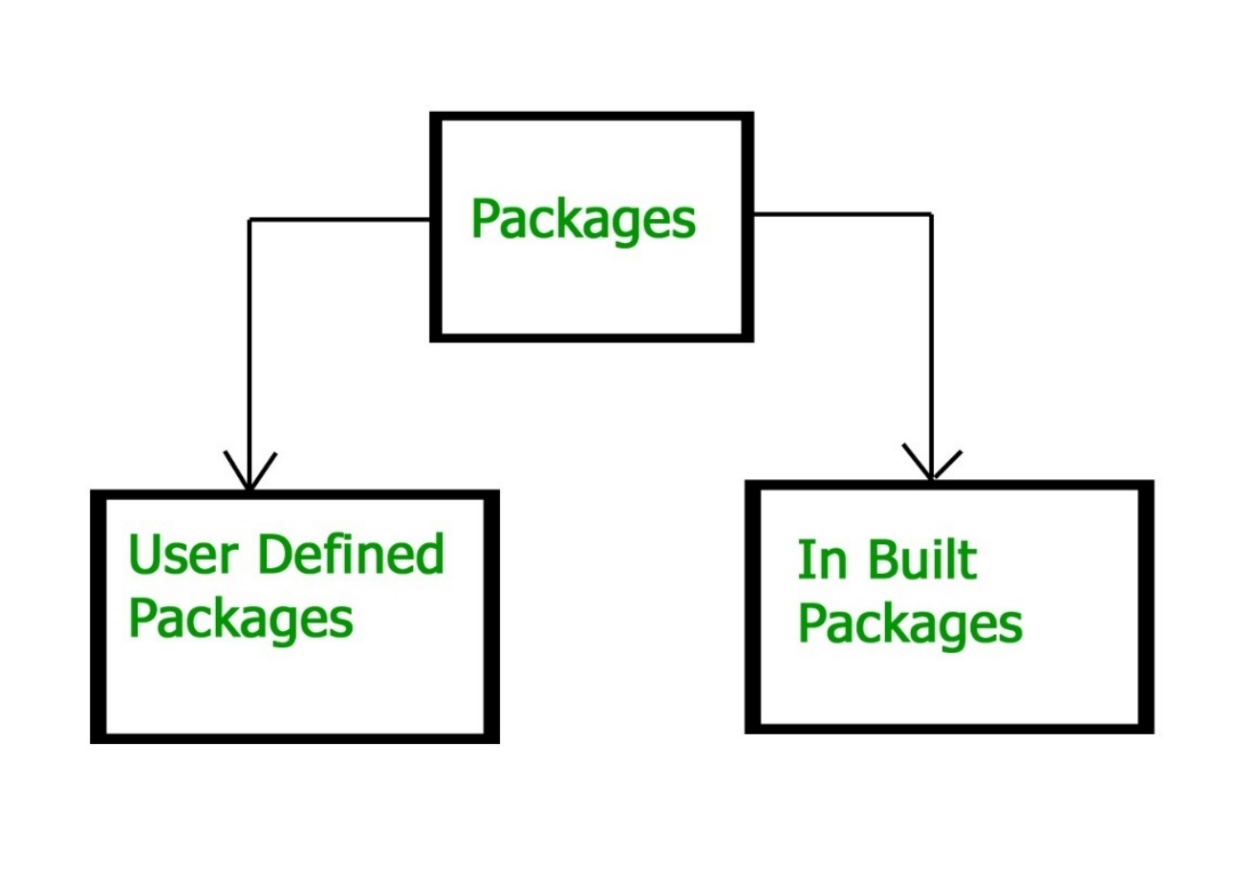
**Package**

A package is nothing but a physical folder structure (directory) that contains a group of related classes, interfaces, and sub-packages according to their functionality. It provides a convenient way to organize your work.

Packages are used for:

* Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
* Making searching/locating and usage of classes and interfaces easier
* Providing controlled access
* Packages can be considered as data encapsulation (or data-hiding).

Packages are divided into two categories:



**Built-in Packages**

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

1. **java.lang:**Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:**Contains classes for supporting input / output operations.
3. **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:**Contains classes for creating Applets.
5. **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc)
6. **java.net:**Contain classes for supporting networking operations.

**User-defined packages**: The package which is defined by the user is called user-defined or custom package in Java. It contains user-defined classes and interfaces.

**Naming Convention for User-defined Package in Realtime Project**

While developing your project, you must follow some naming conventions regarding packages declaration. Packages are named in reverse order of domain names

1. Suppose you are working in IBM and the domain name of IBM is www.ibm.com. You can declare the package by reversing the domain like this:

package com.ibm;

where,

* com ➝ It is generally the company specification name, and the folder starts with com, which is called root folder.
* ibm ➝ Company name where the product is developed. It is the sub folder.

2. niit➝ Client name for which we are developing your product or working on the project.

3. loan ➝ Name of the project.

4. homeloan ➝ It is the name of the modules of the loan project. There are a number of modules in the loan project like a home loan, car loan, or personal loan. Suppose you are working for the Home loan module.

This is a complete packages structure, like a professional which is adopted in the company. Look at another example below:

package com.niit.loan.homeloan.penalty;

## **Best Practices to Create a Package**

While creating your own packages in Java program, keep these points in mind:

* Give a meaningful package name that clearly convey the purpose of the contained classes. It boosts code readability and maintainability.
* Organize classes based on their functionality, that makes it easier to find and work with related classes.
* Always try to avoid over nesting of packages, that can lead to confusion. Always keep a balance between hierarchy and simplicity.
* Follow Java naming conventions. Use the lowercase for package names and follow the reverse domain notation.
* Use proper access modifiers for classes within your package. Limit visibility to only what’s necessary for other classes.

**Package Access**

Package access controls the visibility of classes, methods, and fields within different parts of a program.

### 1. ****Package Declaration****

At the top of a Java source file, you can declare a package like this:

package com.example.myapp;

This declaration places the class in the com.example.myapp package. This package declaration must be the first line in the source file, excluding comments.

### 2. ****Access Modifiers****

Java provides four access levels to control access to classes, methods, and fields:

* **Public:** Accessible from anywhere. For a class, it means the class can be instantiated and used by any other class from any package.

public class MyClass {

public void myMethod() { }

}

* **Protected:** Accessible within the same package and by subclasses (even if they are in different packages).

class MyClass {

protected void myMethod() { }

}

* **Default (Package-Private):** If no access modifier is specified, it is package-private. Accessible only within the same package.

class MyClass {

void myMethod() { }

}

* **Private:** Accessible only within the same class. Not accessible from other classes, even within the same package.

public class MyClass {

private void myMethod() { }

}

**Chapter 9**

**Day 2**

**Inheritance**

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

**Inheritance represents the IS-A relationship which is also known as a parent-child relationship.**

**Terms used in Inheritance**

* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** It allows the reuse of the methods and fields of the existing class when creating a new class.

A subclass can add its own fields and methods. Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as specialization.

Examples of inheritance

|  |  |
| --- | --- |
| Superclass | Subclasses |
| Student | GraduateStudent, UndergraduateStudent |
| Shape | Circle, Triangle, Rectangle, Sphere, Cube |
| Loan | CarLoan, HomeImprovementLoan, MortgageLoan |
| Employee | Faculty, Staff |
| BankAccount | CheckingAccount, SavingsAccount |

Because every subclass object is an object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses. For example, the superclass Vehicle represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass Car represents a smaller, more specific subset of vehicles.

Inheritance relationships form treelike hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses.

**Types of inheritance**

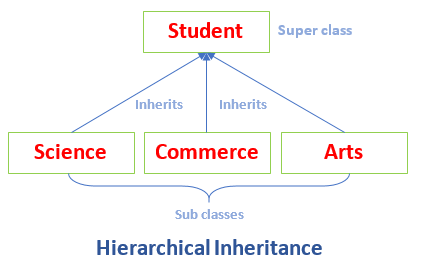
### 1. Single Inheritance

### Single inheritance refers to the scenario where a subclass extends or inherits the properties and behaviors of only one single superclass.

### Inheritance in Java | Real Life Example of Inheritance in Java

### 2. Hierarchical Inheritance

### Hierarchical inheritance allows multiple subclasses to inherit from a single superclass



**Chapter 10**

**Day 3**

**What is Polymorphism?**

Polymorphism is derived from two Greek words, “poly” and “morph”, which mean “many” and “forms”, respectively. Hence, polymorphism meaning in Java refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.

**Types of polymorphism**

* Compile-time polymorphism
* Run-time polymorphism

**1. Compile-time Polymorphism (Method Overloading):**

* This type of polymorphism is resolved during **compile time**.
* It occurs when multiple methods have the same name but differ in the number or type of their parameters.
* Java determines which method to call based on the method signature at compile time.

### ****2. Run-time Polymorphism (Method Overriding)****:

* This type of polymorphism is resolved during **runtime**.
* It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
* The method in the subclass **overrides** the method in the superclass. The method to be invoked is determined at runtime based on the actual object type, not the reference type.

**Abstraction in java**

Abstraction in Java refers to hiding the implementation details of a code and exposing only the necessary information to the user. It provides the ability to simplify complex systems by ignoring irrelevant details and reducing complexity. Java provides many in-built abstractions and few tools to create our own.

Abstraction in Java can be achieved using the following tools it provides

* Abstract classes
* Interfaces

**What is an Abstract Class?**

In Java, an abstract class is a class that cannot be instantiated on its own and typically serves as a blueprint for other classes. Abstract classes are declared using the **abstract** keyword. It can contain abstract methods (methods without a body) as well as concrete methods (methods with a body).

Abstract classes are used when you want to provide a common interface for a group of related classes, but you want to leave some methods to be implemented by the subclasses.

**Final Methods**: When a method is declared as final, it cannot be overridden by subclasses. This is useful for preventing changes to the behaviors of critical methods in a class hierarchy.

**Final Classes**: When a class is declared as final, it means that **no other class can extend (subclass) it**. This is often used to create immutable classes or to ensure that the implementation of a class remains unchanged.

**Points to remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

**Interfaces**

I**nterface** is like a contract that defines a set of methods (functions) that a class must implement, but it doesn’t provide any of the actual code for those methods. Think of it as a blueprint.

**Key Points:**

1. **No Implementation**: An interface only declares methods but does not define how they work.
2. **Multiple Implementations**: Different classes can implement the same interface in different ways.
3. **Supports Polymorphism**: You can treat different classes that implement the same interface as the same type. This is useful for writing flexible and reusable code.

**Key Differences:**

* **Abstract Class**:
  + Can have both abstract and non-abstract methods.
  + Can have member variables (fields).
  + Can provide method implementation.
  + A class can inherit only one abstract class (single inheritance).
* **Interface**:
  + Can only have abstract methods (until Java 8, now it can have default and static methods).
  + Cannot have instance fields.
  + Supports multiple inheritance (a class can implement multiple interfaces).

**Chapter 11**

**Day 4**

**Exception handling**

**What is an Exception?**

**Exception** is an unwanted or unexpected event that occurs during the execution of a program, i.e., at run time. This disrupts the normal flow of the program’s instructions. For example, trying to divide a number by zero or accessing an array element that doesn’t exist are situations that can cause exceptions.

Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

**Major reasons why an exception Occurs**

* **Invalid User Input**: Users may enter data that doesn't match the expected format, causing validation errors.
* **Device Failure**: Hardware malfunctions, such as a hard drive failure or a printer issue, can lead to exceptions.
* **Loss of Network Connection**: Disruptions in network connectivity can result in exceptions during data transmission or remote service calls.
* **Physical Limitations (Out-of-Disk Memory)**: Insufficient resources, like disk space or memory, can lead to application crashes.
* **Code Errors**: Bugs in the code, such as logical errors or incorrect implementations, can trigger exceptions.
* **Out of Bound**: Accessing array or list indices that are beyond the allocated range results in exceptions.
* **Null Reference**: Attempting to use an object that hasn't been initialized leads to null reference exceptions.
* **Type Mismatch**: Trying to perform operations on incompatible data types can cause type-related exceptions.
* **Opening an Unavailable File**: Attempting to access a file that does not exist or is locked can lead to exceptions.
* **Database Errors**: Issues like connection failures, query syntax errors, or data integrity violations can raise exceptions.
* **Arithmetic Errors**: Problems like division by zero or overflow can result in runtime exceptions.

Understanding these reasons can help you in designing robust error handling and prevention mechanisms in your applications. In the programming world, there is a difference between exception and error.

**Errors in java?**

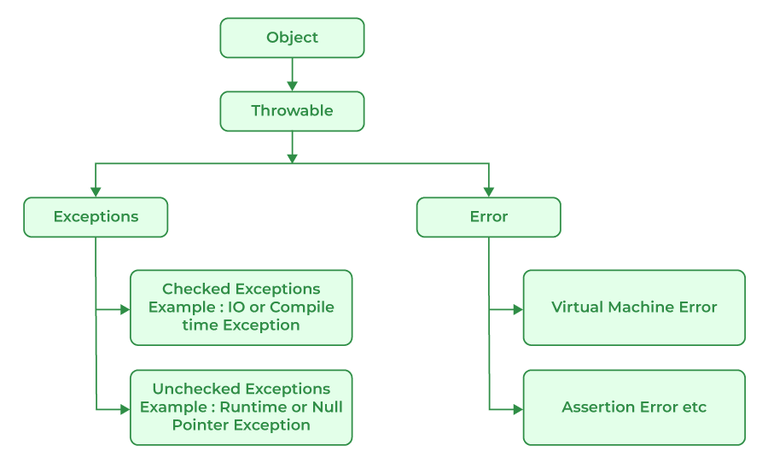
Errors represent serious problems that are typically outside the control of the application or the programmer. They indicate conditions that the application usually cannot recover from. and we should not try to handle errors

### Exception Hierarchy

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. It is divided into two branches.

**Exception**: This class is used for exceptional conditions that user programs should catch. For example, NullPointerException is an example of such an exception.

**Error** is used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). For example, StackOverflowError is an example of such an error.



# **Errors V/s Exceptions**

In Java, errors and exceptions are both types of throwable objects, but they represent different types of problems that can occur during the execution of a program.

Errors are usually caused by serious problems that are outside the control of the program, such as running out of memory or a system crash. Errors are represented by the Error class and its subclasses. Some common examples of errors in Java include:

* **OutOfMemoryError:** Thrown when the Java Virtual Machine (JVM) runs out of memory.
* **StackOverflowError:** Thrown when the call stack overflows due to too many method invocations.
* **NoClassDefFoundError:** Thrown when a required class cannot be found.

Since errors are generally caused by problems that cannot be recovered from, it’s usually not appropriate for a program to catch errors. Instead, the best course of action is usually to log the error and exit the program.

Exceptions, on the other hand, are used to handle errors that can be recovered from within the program. Exceptions are represented by the Exception class and its subclasses. Some common examples of exceptions in Java include:

* **NullPointerException**: Thrown when a null reference is accessed.
* **IllegalArgumentException:** Thrown when an illegal argument is passed to a method.
* **IOException:** Thrown when an I/O operation fails.

Since exceptions can be caught and handled within a program, it is common to include code to catch and handle exceptions in Java programs. By handling exceptions, you can provide more informative error messages to users and prevent the program from crashing.

## **Types of Exceptions**

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

### ****1. User-Defined or custom Exceptions:****

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

**Methods to print the Exception information:**

**1 printStackTrace()**: This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.

**2. toString()**: The toString() method prints exception information in the format of the Name of the exception: description of the exception.

**3. getMessage()** : The getMessage() method prints only the description of the exception.

### ****2. Built-in Exceptions****

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. They are divided into two parts.

* **Checked Exceptions:**Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
* **Unchecked Exceptions: Unchecked exception** are exceptions that are thrown at the run time (and therefore, also known as **Runtime Exceptions**). They can be ignored at the time of compiling. Examples of Unchecked Exceptions would be ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, etc.

**How to Handle Exceptions in Java**

Exceptions can be handled using **`try`, `catch`,** and optionally **`finally`** blocks.

**Try Block**

The **`try`** block contains the code that might cause an exception. You write the code that you want to monitor for errors in the try block.

**Catch Block**

The **`catch`** block catches and handles the exception if one occurs in the `try` block. You can have multiple `catch` blocks to handle different types of exceptions.

**Finally Block**

The `finally` block contains code that will always run, regardless of whether an exception was thrown or not. It is commonly used for cleaning up resources like closing files or database connections.

**Using throw and throws keyword in Java**

**Java throw**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

## **Java throws**

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw an exceptions. The caller to these methods has to handle the exception using a try-catch block.

**What is an Uncaught Exception?**

An uncaught exception occurs when an error happens in your program and there isn’t any code to handle it (i.e., no catch block for that exception). When this happens, Java typically prints an error message and a stack trace, which shows you where the error occurred.

**Uncaught Exception When Using Single-Threaded vs. Multi-Threaded Programs**

* **Single-Threaded**: If your program runs in a single thread (one sequence of execution), an uncaught exception will cause the entire program to stop running immediately. You’ll see the error message, and nothing else will execute.
* **Multi-Threaded**: If your program uses multiple threads (think of them as separate lines of execution that can run concurrently), an uncaught exception will only terminate the thread in which the exception occurred. Other threads can continue running. However, this can lead to problems if one thread relies on the results of another thread that has terminated due to the exception.

**Example**

Imagine you have a multi-threaded program where:

* Thread A processes data.
* Thread B processes the results of Thread A.

If Thread A encounters an uncaught exception and terminates, Thread B may be left trying to work with incomplete or unavailable data, which could lead to further errors or unexpected behavior.

**When to Use Exception Handling**

1. **Synchronous Errors**: Use exception handling to manage errors that arise during the execution of code, such as:
   * **Out-of-Range Access**: Accessing an array with an invalid index.
   * **Arithmetic Errors**: Handling cases like division by zero or overflow.
   * **Invalid Input**: When method parameters do not meet expected criteria.
2. **Resource Management**: Use it to ensure that resources (like file handles or database connections) are properly released, even in the case of an error.
3. **Code Clarity**: Implement exception handling to separate error-handling logic from regular code flow, making your code easier to read and maintain.
4. **Logging and Debugging**: Capture exceptions to log errors or provide debugging information without crashing the application.

**When Not to Use Exception Handling**

1. **Asynchronous Events**: Do not rely on exceptions for handling asynchronous operations like I/O completions or UI events (e.g., mouse clicks
2. **Control Flow**: Avoid using exceptions for normal control flow, as this can lead to performance issues and reduced readability.
3. **Non-recoverable Errors**: If an error occurs that you cannot recover from (like a fatal application error), consider logging and shutting down gracefully rather than trying to handle it in the usual flow.

**Best Practices**

* **Specific Exceptions**: Catch specific exceptions rather than a general one to avoid hiding bugs.
* **Resource Cleanup**: Use finally blocks or try-with-resources (in languages like Java) for proper resource management.
* **Avoid Silent Failures**: Always handle exceptions in a way that provides feedback, either through logging or user notifications.

By following these guidelines, you can effectively manage errors in your applications while maintaining code clarity and performance.

**Assertions**

Assertions are statements in your code that help you verify whether certain conditions are true while your program is running. They serve as checks that can catch errors early in the development process.

### Java's Assert Statement

Java provides a built-in way to implement assertions through the assert statement. There are two forms:

1. **Basic Form**:

assert expression;

This checks if expression is true. If it's false, an AssertionError is thrown, which halts the program.

1. **Detailed Form**:

assert expression1 : expression2;

This checks expression1. If it's false, it throws an AssertionError with a message specified by expression2. This message can provide more context about the error, making it easier to debug.

Users shouldn’t encounter AssertionErrors—these should be used only during program development. For this reason, you shouldn’t catch AssertionErrors. Instread, allow the program to terminate, so you can see the error message, then locate and fix the source of the problem. You should not use assert to indicate runtime problems in production code instead use the exception mechanism for this purpose

To enable assertions when running the program, you would use the -ea flag like this:

java -ea AssertTest

This approach is useful during development and testing but should generally be disabled in production environments to avoid performance overhead and to prevent exposing internal logic to end users. You use assertions primarily for debugging and identifying logic errors in an application. You must explicitly enable assertions when executing a program, because they reduce assert expression

How to enable assertions in NetBeans, follow these steps:

1. **Open your Project**: Start by opening your project in NetBeans.
2. **Project Properties**: Right-click on your project in the Projects window and select **"Properties."**
3. **Run Configuration**: In the Project Properties dialog, select **"Run"** from the list on the left.
4. **VM Options**: In the **"VM Options"** field, add the following line:

-ea

1. **Apply Changes**: Click **"OK"** to save the changes.

Now, when you run your program, assertions will be enabled, and any assertion failures will trigger an AssertionError as expected. To disable assertion, follow the same steps and type ‘-da ‘.

### Types of Assertions

1. **Preconditions**: These are assertions that check the state of the program before a method is called. They ensure that the method is being used correctly. For example, if a method expects a number to be positive, a precondition assertion would verify this before the method runs.
2. **Postconditions**: These assertions check the state after a method has completed. They confirm that the method worked as intended. For example, if a method is supposed to return a sorted list, a postcondition assertion would check if the list is indeed sorted after the method runs.

**Week 6**

**Chapter 12**

**Day 1**

**GUI: components and basic event handling**

A Graphical User Interface (GUI) is designed to make it easier for users to interact with applications by providing a visual way to control them. Instead of typing commands, users can click buttons, drag sliders, or select options from menus, which makes using software more intuitive.

A GUI (pronounced “GOO-ee”) gives an application a distinctive “look and-feel.” GUIs are built from GUI components. These are sometimes called controls or widgets (short for window gadgets).

**GUI Components**:

A **GUI component** refers to any visual element of a **Graphical User Interface (GUI)** that users can interact with. These components allow users to input, view, or manipulate data, and they typically appear as objects in code. For example, if you're building a desktop application, you may have buttons, text boxes, sliders, labels, and other interactive elements. These are all GUI components.

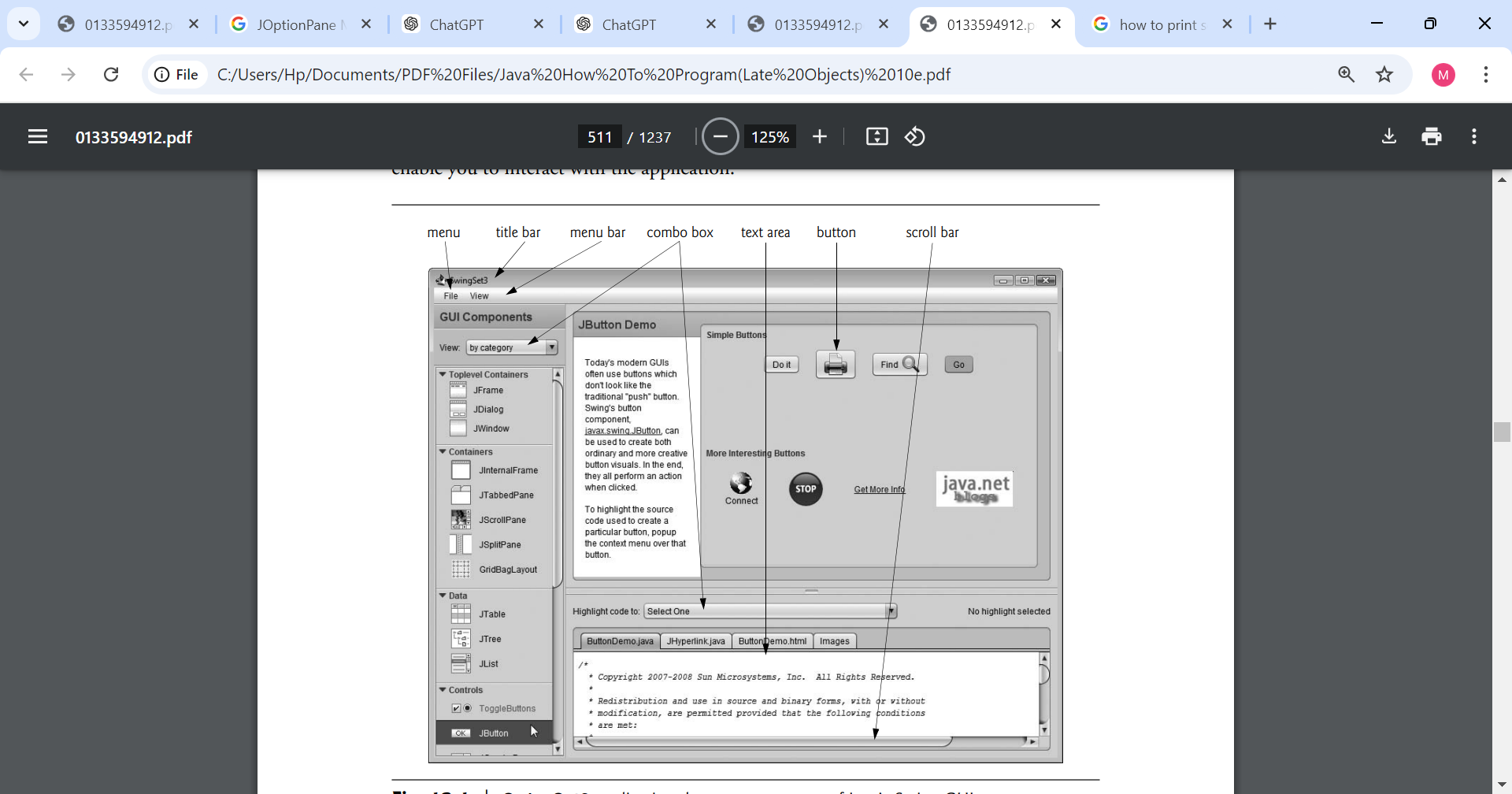
**GUI component as an Object**: In object-oriented programming (OOP), everything is an object, and each object has properties (like size, color, text) and behaviors (like clicking, dragging, etc.). So, when we say "A GUI component is an object," we mean that these GUI elements (like buttons, text fields, etc.) are represented as objects in code.

**IDE Support for GUI Design**

1. **Visual Design Tools**: Many Integrated Development Environments (IDEs) offer graphical tools that allow developers to design user interfaces visually. This means you can:
   * **Drag and Drop**: Place components like buttons, text fields, and labels directly onto a design canvas.
   * **Adjust Properties**: Specify size, location, and other attributes using mouse actions or property panels.
2. **Automatic Code Generation**: When you use these visual tools, the IDE automatically generates the underlying code for the GUI. This can save significant time and effort, as you don’t have to manually write out the layout and configuration.
3. **Variability Among IDEs**: Different IDEs have different ways of generating and managing GUI code. This means that the code produced by one IDE may not be compatible with another, and the way you interact with components can vary.
4. **Learning by Hand-Coding**: While IDEs can simplify GUI creation, the chapter emphasizes writing GUI code by hand. This approach helps deepen your understanding of how GUIs work under the hood and can improve your coding skills.
5. **Encouragement to Experiment**: The text encourages you to use your preferred IDE(s) to visually build GUIs. This hands-on experience can help reinforce concepts and familiarize you with the specific tools available in your chosen environment.

**SwingSet3 Demo Application**

The **SwingSet3 Demo Application** is a great illustration of how Java's Swing components work together to create a functional and interactive GUI.



### Java’s Nimbus Look-and-Feel

The **look-and-feel** of a GUI refers to its visual appearance and the way users interact with it. In Java, the Nimbus look-and-feel is a modern, cross-platform design that provides a sleek and consistent interface for Swing applications. Here’s how you can implement and configure Nimbus in your Java applications:

**Look-and-Feel Definition**:

* **Look**: Visual aspects, including colors, fonts, and styles.
* **Feel**: Functional components, like buttons, menus, and how users interact with them.

**Nimbus Look-and-Feel**:

Swing has a cross-platform look-and-feel known as Nimbus which Provides a contemporary design for Swing applications, enhancing user experience with a polished appearance.

**How to Use Nimbus:**

There are three main ways to set Nimbus as the look-and-feel for your Java applications:

1. **Set as Default for All Java Applications**:
   * Create a file named swing.properties in the lib folder of both your JDK and JRE installation directories.
   * Add the following line to the swing.properties file:

nimbus=javax.swing.plaf.nimbus.NimbusLookAndFeel

* + If your IDE uses a nested JRE within the JDK, ensure the swing.properties file is also placed in that lib folder.

1. **Set at Launch via Command-Line Argument**:
   * You can specify Nimbus when launching your application by adding the following command-line argument:

-Dswing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel

* + This should be placed after the java command and before the application name.

1. **Set Programmatically in Your Application**:
   * You can also set Nimbus in your Java code. Here's an example of how to do this:

try {

UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");

} catch (Exception e) {

e.printStackTrace();

}

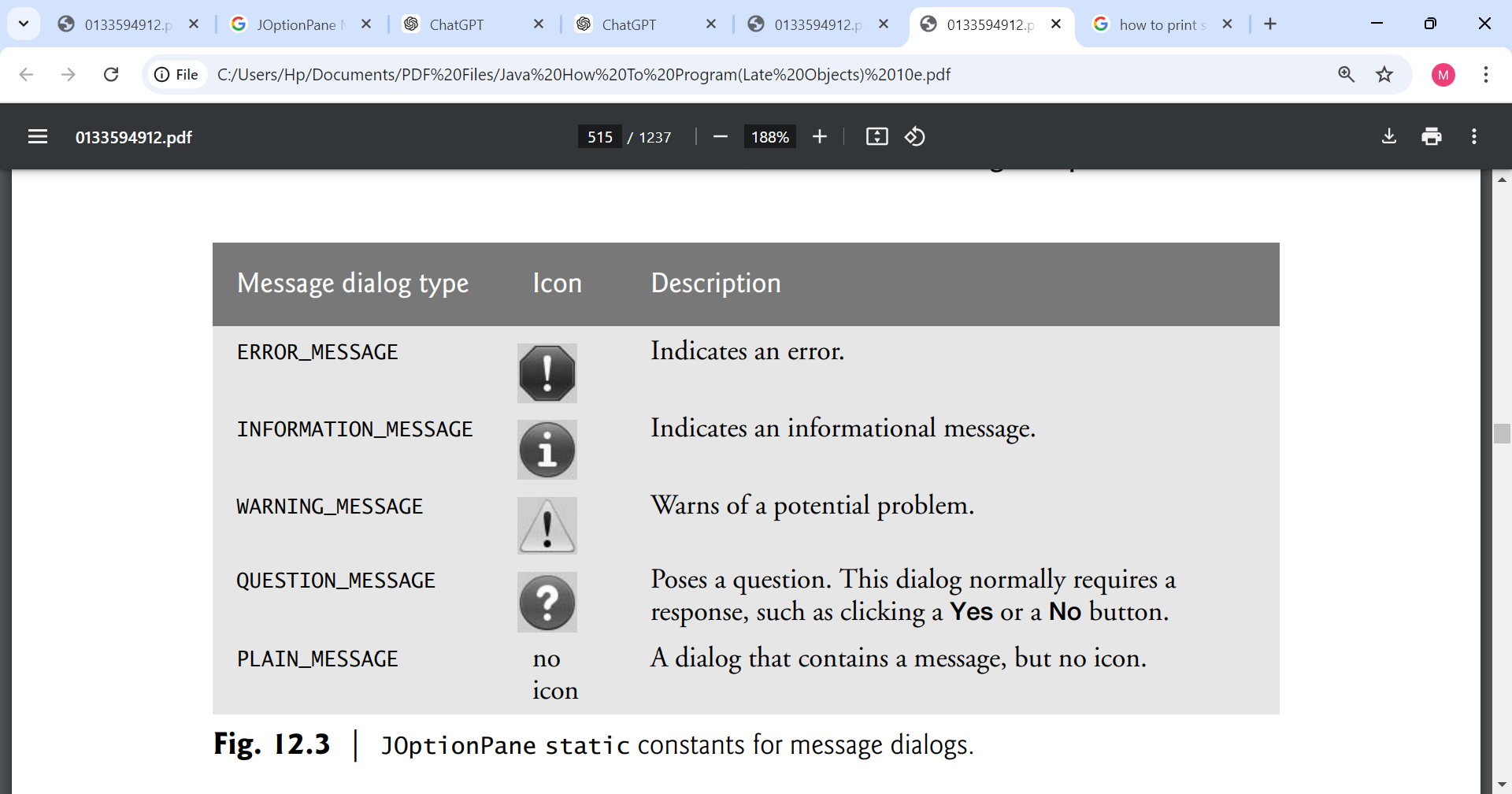
* + This code should be executed before creating any GUI components to ensure that the look-and-feel is applied.

### Simple GUI-Based Input/Output with JOptionPane

In Java, user interaction is often handled through dialog boxes rather than command-line input and output, providing a more intuitive experience. The **JOptionPane** class in the javax.swing package is specifically designed for this purpose, offering a variety of prebuilt dialog boxes for both obtaining input and displaying output.

**Input Dialogs**: These dialog boxes allow you to prompt the user for information. For instance, you can ask the user to enter values such as names or numbers.

**Message Dialogs**: These are used to display information, such as results or alerts, to the user in a pop-up window.



**Swing vs. AWT**

In Java, GUI components can be created using two primary libraries: the Abstract Window Toolkit (AWT) and Swing.

**AWT (Abstract Window Toolkit) in package java.awt.**

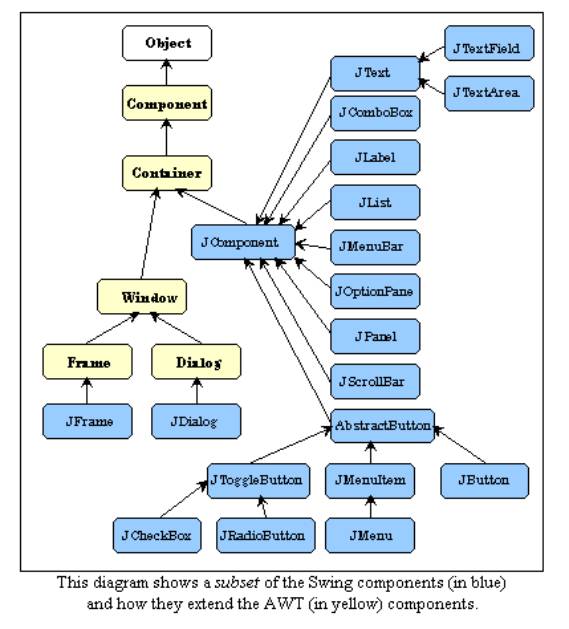
1. **Native Look and Feel**:
   * AWT components are heavyweight, meaning they rely on the native operating system’s GUI components. This makes them look and behave like standard components of the host OS. For instance, a button created with AWT on Windows will have the same appearance as other Windows buttons.
2. **Platform Dependency**:
   * Because AWT uses the underlying OS components, its behavior and appearance can differ from one platform to another (Windows, macOS, Linux). This can lead to inconsistencies in how applications look and feel across different environments.
3. **Limited Component Set**:
   * AWT provides a basic set of components, such as buttons, labels, text fields, and panels. However, it lacks some more advanced components and features that are often required for modern applications.
4. **Event Handling**:
   * AWT uses a simple event model for handling user interactions, which can sometimes be less flexible compared to Swing.

**Swing in package javax.swing**

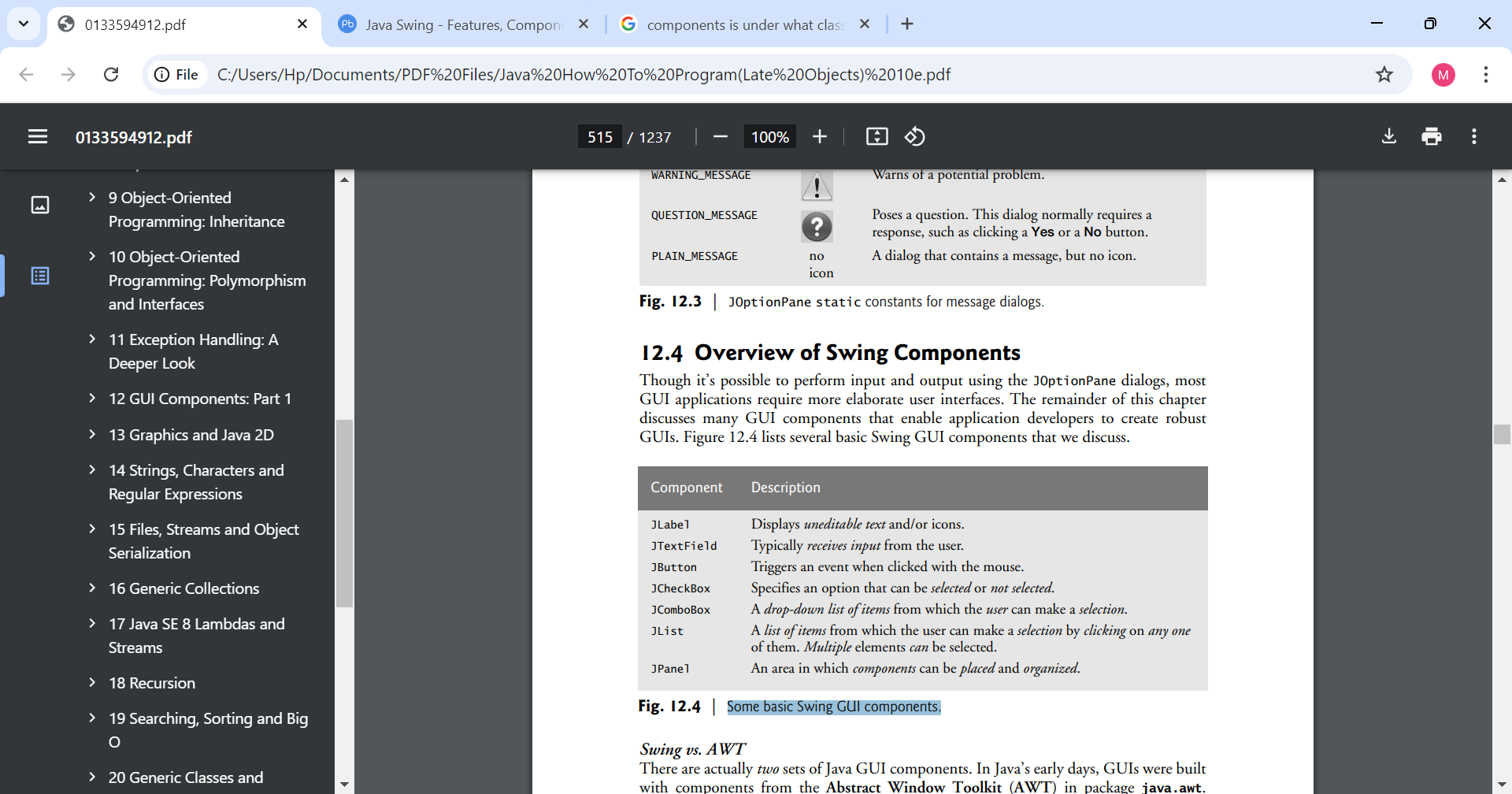
1. **Lightweight Components**:
   * Swing components are lightweight, meaning they are written entirely in Java and do not rely on native GUI components. This allows for a consistent look and feel across all platforms, making it easier to develop cross-platform applications.
2. **Customizable Look and Feel**:
   * Swing allows developers to customize the appearance of components extensively. You can apply different "look-and-feel" themes to change how your application looks, independent of the platform. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel.
3. **Rich Component Set**:
   * Swing provides a more extensive and sophisticated set of components, including advanced features like trees, tables, and text panes. This enables the development of more complex user interfaces.
4. **More Powerful Event Handling**:
   * Swing employs a more sophisticated event handling model, allowing for better control over user interactions and responsiveness in the GUI.

**Java Swing Class Hierarchy**

The Java Swing API hierarchy is shown below:



**Some common swing components and their functions**

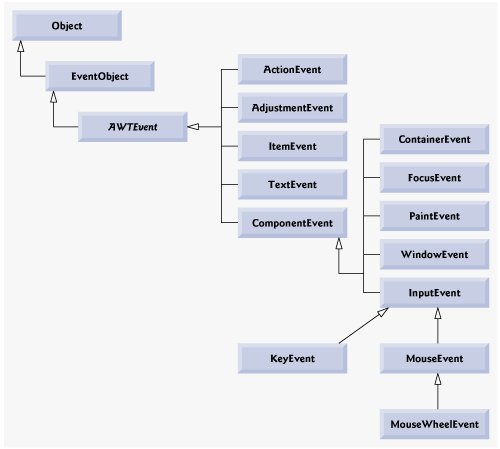


**What is an Event and Event Handler?**

* **Events:** GUIs are event driven. When a user interacts with a GUI component and it drives the program to perform a task, the interaction is known as an event. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses is called an event. Some common user interactions that cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse.
* **Event Handlers**: These are blocks of code that respond to events. When an event occurs, the corresponding event handler executes to perform a task. The process of responding to events is known as event handling.

**Common GUI Event Types and Event Listener Interfaces**

|  |  |  |
| --- | --- | --- |
| **Actions to trigger event** | **Event classes** | **Listeners interfaces** |
| * **Button Clicks**: When a user clicks a JButton. * **Menu Item Selection**: When a user selects a JMenuItem from a JMenu. * **Text Field Submission**: When user presses Enter in a JTextField * **Combo Box Selection**: When a user selects an item from a JComboBox. | ActionEvent | ActionListener |
| * **Mouse Clicked: Triggered** when a mouse button is clicked (pressed and released). * **Mouse Pressed:** Triggered when a mouse button is pressed down. * **Mouse Released:** Triggered when a mouse button is released. * **Mouse Entered:** Triggered when the mouse cursor enters the component's area. * **Mouse Exited:** Triggered when the mouse cursor exits the component's area. * **Mouse Moved:** Triggered when the mouse is moved within the component's area. * **Mouse Dragged:** Triggered when the mouse is moved while a button is pressed. | MouseEvent | MouseListener and MouseMotionListener |
| * Key Pressed: Triggered when a key is pressed down. * Key Released: Triggered when a key is released. * Key Typed: Triggered when a key is pressed and released (i.e., when a character is typed). | KeyEvent | KeyListener |
| * Item Selected: Triggered when an item is selected (e.g., when a checkbox is checked or a radio button is chosen). * Item Deselected: Triggered when an item is deselected (e.g., when a checkbox is unchecked | ItemEvent | ItemListenerk |
| * **TextEvent**: This event is generated when the text in a text component (like a text field or text area) is changed. It’s useful for monitoring user input. | TextEvent | TextListener |
| * **AdjustmentEvent**: This event is related to adjustments made to components like scroll bars and sliders. It’s useful for responding to changes in the value of these components. | AdjustmentEvent | AdjustmentListener |
| * **ItemEvent**: This event is triggered when an item (such as a checkbox or a radio button) is selected or deselected. It’s often used with item listeners to handle selection changes. | ItemEvent | ItemListener |
| * Opening * Closing * Minimizing * resizing. | WindowEvent | WindowListener |
|  | FocusEvent | FocusListener |

****

Control/action Listener interface

----------------- ---------------------

Button click ActionListener

Scroll Bar AdjustmentListener

Slider ChangeListener

Text Field focus FocusListener

Check Box ItemListener

Text entered KeyListener

Mouse click MouseListener

Mouse wheel MouseWheelListener

Window closes WindowListener

**\*\*Chapter 13**

Day 2

**Layout Managers**

The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout

**BorderLayout**

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

* **BorderLayout():** creates a border layout but with no gaps between the components.
* **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

# **GridLayout**

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

### Constructors of GridLayout class

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

# **FlowLayout**

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

### Fields of FlowLayout class

1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

### Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

# **BoxLayout**

The **Java BoxLayout class** is used to arrange the components either vertically or horizontally. For this purpose, the BoxLayout class provides four constants. They are as follows:

#### **Note: The BoxLayout class is found in javax.swing package.**

### Fields of BoxLayout Class

1. **public static final int X\_AXIS:** Alignment of the components are horizontal from left to right.
2. **public static final int Y\_AXIS:** Alignment of the components are vertical from top to bottom.
3. **public static final int LINE\_AXIS:** Alignment of the components is similar to the way words are aligned in a line, which is based on the ComponentOrientation property of the container. If the ComponentOrientation property of the container is horizontal, then the components are aligned horizontally; otherwise, the components are aligned vertically. For horizontal orientations, we have two cases: left to right, and right to left. If the value ComponentOrientation property of the container is from left to right, then the components are rendered from left to right, and for right to left, the rendering of components is also from right to left. In the case of vertical orientations, the components are always rendered from top to bottom.
4. **public static final int PAGE\_AXIS:** Alignment of the components is similar to the way text lines are put on a page, which is based on the ComponentOrientation property of the container. If the ComponentOrientation property of the container is horizontal, then components are aligned vertically; otherwise, the components are aligned horizontally. For horizontal orientations, we have two cases: left to right, and right to left. If the value ComponentOrientation property of the container is also from left to right, then the components are rendered from left to right, and for right to left, the rendering of components is from right to left. In the case of vertical orientations, the components are always rendered from top to bottom.

### Constructor of BoxLayout class

1. **BoxLayout(Container c, int axis):** creates a box layout that arranges the components with the given axis.

# **CardLayout**

The **Java CardLayout** class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

### Constructors of CardLayout Class

1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

### Commonly Used Methods of CardLayout Class

* **public void next(Container parent):** is used to flip to the next card of the given container.
* **public void previous(Container parent):** is used to flip to the previous card of the given container.
* **public void first(Container parent):** is used to flip to the first card of the given container.
* **public void last(Container parent):** is used to flip to the last card of the given container.
* **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

Day 3

• GUI: event handling and adapter classes

**Chapter 14**

**Characters, Strings, and Regular Expressions**

Day 4

**Introduction**

This chapter introduces Java’s capabilities for string and character processing, which are essential for validating input, displaying information, and various text manipulations. It highlights the use of classes such as String, StringBuilder, and Character from the java.lang package, forming the basis for string and character manipulation in Java. Additionally, the chapter covers regular expressions, emphasizing their role in input validation and their implementation through the String, Matcher, and Pattern classes in the java.util.regex package. Overall, these tools are useful for developing applications like text editors and word processors.

**Fundamentals of Characters and Strings**

Characters are the fundamental building blocks of Java source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—are interpreted by the Java compiler as a series of instructions used to accomplish a task. A program may contain character literals.

**Character Literals**

A character literal is an integer value represented as a character in single quotes. For example, 'z' represents the integer value of z, and '\t' represents the integer value of a tab character. The value of a character literal is the integer value of the character in the Unicode character set.

**Class Character**

The Character class is a wrapper for the primitive data type char. It provides methods to manipulate and convert characters.

* + **Static Methods**:
    - Character.isDigit(char ch): Checks if the character is a digit.
    - Character.isLetter(char ch): Checks if the character is a letter.
    - Character.toUpperCase(char ch): Converts a character to uppercase.
    - Character.toLowerCase(char ch): Converts a character to lowercase.

**Strings**

String is a sequence of characters treated as a single unit. A string may include letters, digits and various special characters, such as +, -, \*, / and $. A string is an object of class String. String literals (stored in memory as String objects) are written as a sequence of characters in double quotation marks. For example

"John Q. Doe" (a name)

"9999 Main Street" (a street address)

"Waltham, Massachusetts" (a city and state)

"(201) 555-1212" (a telephone number)

A string may be assigned to a String reference. The declaration String color = "blue"; initializes String variable color to refer to a String object that contains the string "blue"

String color = "blue";

This statement does three things:

1. Declares a String reference → String color: This means color is a variable that can hold a reference (memory address) to a String object.
2. Creates a String object containing "blue": Since "blue" is a String literal, Java automatically stores it in the String Pool (inside heap memory).
3. Assigns the reference to the color variable: The variable color now refers to (or "points to") the "blue" object in memory.

**Class String**

Class String is used to represent strings in Java.

* **Immutability**: Once a String object is created, it cannot be changed. Any modification results in a new String object.
* **String Literals**: Strings can be created using double quotes, e.g., "Hello, World!".

**String Constructors**

Class String provides constructors for initializing String objects in a variety of ways.

* String(): Creates an empty string.
* String(String str): Creates a string from another string.
* String(char[] charArray): Creates a string from a character array.

**Methods used for string manipulation**

In Java, the String class provides several methods for manipulating strings. Here are some common methods:

1. **length()**: Returns the length of the string.

String str = "Hello, World!";

System.out.println(str.length());

1. **charAt(int index)**: Returns the character at the specified index.

String str = "Java";

System.out.println(str.charAt(1));

1. **substring(int beginIndex)**: Returns a substring starting from the specified index.

String str = "Programming";

System.out.println(str.substring(3));

1. **substring(int beginIndex, int endIndex)**: Returns a substring from beginIndex to endIndex - 1.

String str = "Java Programming";

System.out.println(str.substring(5, 16));

1. **indexOf(String str)**: Returns the index of the first occurrence of the specified substring.

String str = "I love Java!";

System.out.println(str.indexOf("Java"));

System.out.println(str.indexOf("Python"));

1. **toLowerCase()**: Converts the string to lower case.

String str = "HELLO";

System.out.println(str.toLowerCase());

1. **toUpperCase()**: Converts the string to upper case.

String str = "hello";

System.out.println(str.toUpperCase());

1. **trim()**: Removes leading and trailing whitespace.

String str = " Java ";

System.out.println(str.trim());

1. **replace(char oldChar, char newChar)**: Replaces all occurrences of a specified character with a new character.

String str = "banana";

System.out.println(str.replace('a', 'o'));

1. **split(String regex)**: Splits the string into an array based on the specified based on a **delimiter**.

String sentence = "Java is fun";

String[] words = sentence.split(" ");

**Bonus Challenge: Using Multiple String Methods Together**

String sentence = " Java is AWESOME! "; String result = sentence.trim().toLowerCase().replace("awesome", "great");

System.out.println(result);

**String tokens**

In Java, string tokens refer to individual elements or substrings obtained from a larger string when it is split based on specified delimiters. The process of breaking a string into tokens is commonly done using:

### Using split() Method: The split(String regex) method is a simple and powerful way to tokenize a string based on a regular expression.

### Using StringTokenizer Class: The StringTokenizer class provides an alternative way to tokenize a string. It's part of the java.util package.

**Class StringBuilder**

* StringBuilder is a class used to create and manipulate mutable (modifiable) strings. Unlike String, which is immutable, StringBuilder allows you to modify the string without creating a new object every time, making it more efficient for operations like concatenation, insertion, deletion, and replacement. Ideal for scenarios where you need to manipulate strings frequently, such as constructing strings in loops.
* **Common Methods**:

### append(String str)**: Adds text at the end**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World!");

### insert(int offset, String str)**: Inserts text at a given position**

StringBuilder sb = new StringBuilder("Hello");

sb.insert(5, " Java");

### replace(int start, int end, String str)**: Replaces part of the string**

StringBuilder sb = new StringBuilder("Hello World");

sb.replace(6, 11, "Java");

### delete(int start, int end)**: Deletes a part of the string**

StringBuilder sb = new StringBuilder("Hello World");

sb.delete(5, 11);

### reverse()**: Reverses the string**

StringBuilder sb = new StringBuilder("Java");

sb.reverse();

### capacity()**: Returns the capacity of the StringBuilder**

StringBuilder sb = new StringBuilder();

System.out.println(sb.capacity()); // Default: 16

sb.append("Hello");

System.out.println(sb.capacity()); // Still 16, unless exceeded

### ensureCapacity(int minCapacity) **:Ensures the capacity is at least** minCapacity

StringBuilder sb = new StringBuilder();

sb.ensureCapacity(50); // Ensures capacity is at least 50

### charAt(int index) ****:**Returns the character at a given index**

StringBuilder sb = new StringBuilder("Java");

char ch = sb.charAt(2);

### setCharAt(int index, char ch): **Modifies a character at a given index**

StringBuilder sb = new StringBuilder("Java");

sb.setCharAt(0, 'Y');

### length()**: Returns the length of the string**

StringBuilder sb = new StringBuilder("Hello");

System.out.println(sb.length());

**Class StringBuffer**

Similar to StringBuilder, StringBuffer is also used to create mutable strings but is synchronized, making it thread-safe. Use StringBuffer when working in a multi-threaded context where multiple threads might modify the same string.

* **Common Methods**: Shares many methods with StringBuilder, such as

append(), insert(), delete(), and toString().

**Using the append method in both StringBuilder and StringBuffer**

The append method is a crucial feature of both the StringBuilder and StringBuffer classes. It allows you to add data to the end of an existing string.

1. **Appending Strings**:
2. **Appending Different Data Types**: The append method can take various data types, such as:
   * **String**
   * **char**
   * **int**
   * **float**
   * **double**
   * **boolean**
   * **char arrays**
3. **Chaining**: The append method returns a reference to the same object, allowing for method chaining:
4. **Performance**:
   * **StringBuilder**: Designed for use in a single-threaded context, it is more efficient when appending strings frequently.
   * **StringBuffer**: While it also provides the append method, it is synchronized, making it thread-safe but potentially slower in performance due to the overhead of synchronization.

### StringBuilder Insertion and Deletion Methods

StringBuilder offers various methods to insert and delete characters or substrings, making it a flexible tool for manipulating strings. Here's a closer look at these methods:

**Insertion Methods**

1. **insert(int offset, String str):** Inserts the specified string at the given offset.
2. **insert(int offset, char ch):** Inserts a single character at the specified offset.

**insert(int offset, int value):** Inserts the string representation of an integer at the specified offset.

**insert(int offset, boolean value):** Inserts the string representation of a boolean value.

**insert(int offset, char[] chars, int start, int length):** Inserts a portion of a character array into the StringBuilder.

**Deletion Methods**

**delete(int start, int end):** Deletes the substring that begins at the specified start index and extends to the character at end - 1.

**deleteCharAt(int index):** Deletes the character at the specified index.

**Methods used for comparing of strings**

**==**:

Checks if two references point to the same object in memory. Not recommended for comparing string content; use it for checking if two references are the same instance.

**equals()**:

Compares the content of two strings for equality. Use this when you need to check if two strings have the same sequence of characters ( it is case-sensitive).

**equalsIgnoreCase()**:

Compares the content of two strings for equality while ignoring case differences. Use this when the case of characters should not affect the comparison.

**compareTo()**:

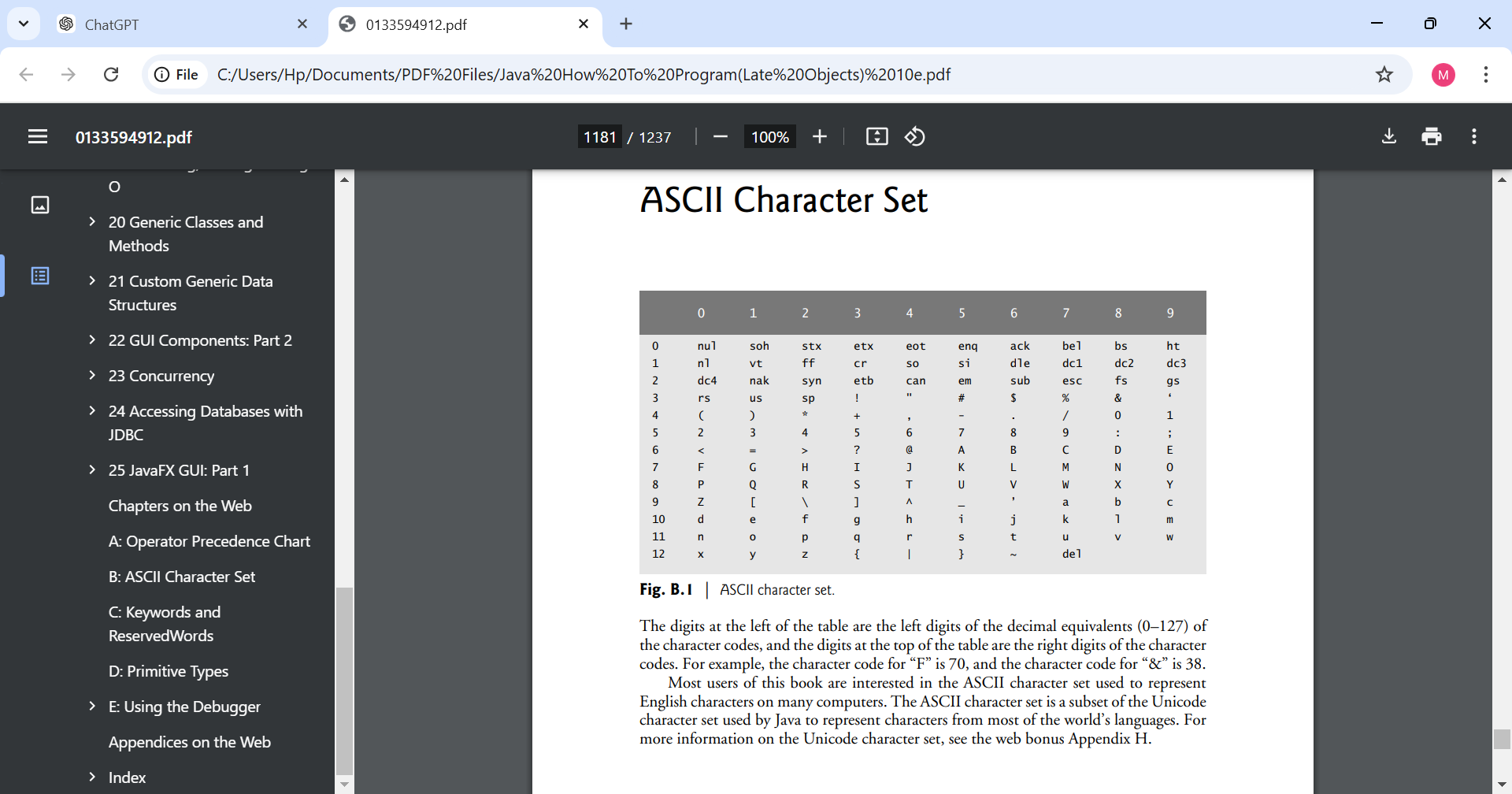
Compares two strings lexicographically. Use this when you need to determine the order of strings (i.e., sorting).

**Lexicographical Order**: This is similar to dictionary order. Strings are compared based on the Unicode value of each character in the strings.

**Return Values**:

* Returns a negative integer if the calling string is lexicographically less than the argument string.
* Returns zero if the two strings are equal.
* Returns a positive integer if the calling string is lexicographically greater than the argument string.

**Case Sensitivity**: The comparison is case-sensitive, meaning uppercase letters are considered less than lowercase letters.



**regionMatches()**:

Compares a specific segment of one string to a segment of another string. Use this to check if parts of two strings match (case-sensitive or case-insensitive).

**startsWith() and endsWith()**:

Checks if a string starts or ends with a specific substring. Use these methods to validate prefixes or suffixes in strings.

* + **Example**:

**String index Methods:**

1. **indexOf(char ch)**
   * **Purpose**: Finds the first occurrence of a specified character in the string.
   * **Returns**: The index of the character, or -1 if the character is not found.
   * **Example**: letters.indexOf('c'); // Finds the index of 'c'
2. **indexOf(char ch, int fromIndex)**
   * **Purpose**: Finds the first occurrence of a specified character starting the search at a specified index.
   * **Returns**: The index of the character, or -1 if the character is not found after the specified index.
3. **indexOf(String str)**
   * **Purpose**: Finds the first occurrence of a specified substring in the string.
   * **Returns**: The index of the substring, or -1 if it is not found.
4. **indexOf(String str, int fromIndex)**
   * **Purpose**: Finds the first occurrence of a specified substring starting the search at a specified index.
   * **Returns**: The index of the substring, or -1 if it is not found after the specified index.
5. **lastIndexOf(char ch)**
   * **Purpose**: Finds the last occurrence of a specified character in the string.
   * **Returns**: The index of the character, or -1 if it is not found.
6. **lastIndexOf(char ch, int fromIndex)**
   * **Purpose**: Finds the last occurrence of a specified character starting the search at a specified index.
   * **Returns**: The index of the character, or -1 if it is not found before the specified index.
7. **lastIndexOf(String str)**
   * **Purpose**: Finds the last occurrence of a specified substring in the string.
   * **Returns**: The index of the substring, or -1 if it is not found.
8. **lastIndexOf(String str, int fromIndex)**
   * **Purpose**: Finds the last occurrence of a specified substring starting the search at a specified index.
   * **Returns**: The index of the substring, or -1 if it is not found before the specified index.

**String Value Methods**

As we’ve seen, every object in Java has a toString method that enables a program to obtain the object’s string representation. Unfortunately, this technique cannot be used with primitive types because they do not have methods. Class String provides static methods that take an argument of any type and convert it to a String object.

1. **String.valueOf(char[] data)**: Converts a character array to a string.
2. **String.valueOf(char[] data, int offset, int length)** : Converts a portion of a character array to a string.
3. **String.valueOf(boolean b)**: Converts a boolean value to its string representation ("true" or "false").
4. **String.valueOf(char c)**: Converts a single character to its string representation.
5. **String.valueOf(int i)**: Converts an integer to its string representation.
6. **String.valueOf(long l)**: Converts a long value to its string representation.
7. **String.valueOf(float f)**: Converts a float value to its string representation.
8. **String.valueOf(double d)**: Converts a double value to its string representation.
9. **String.valueOf(Object obj)**: Converts an object to its string representation by calling the object's toString() method.

**Week 7**

**Day 1**

## **Regular expressions**

Regular expressions (**regex**) are used to search, match, and manipulate strings based on patterns. Java provides the java.util.regex package, which includes Pattern and Matcher classes to work with regex. Regex can be useful tool in form validation, searching text, and data parsing.

## **How to use Regex**

* **Imports**: Regex in Java requires importing java.util.regex.\*. it contains three classes. They are:
* **Pattern, Matcher and PatternSyntaxExceptionClasses**:
  + **Pattern**:

Pattern is a class that represents a compiled representation of a regular expression. When you create a Pattern, the regex is compiled into a form that can be used for efficient matching.

To create a Pattern, you use Pattern.compile(String regex). This compiles the provided regex string into a Pattern object.

* + **Matcher**:

Matcher is a class that performs matching operations against an input string using a Pattern. It provides methods to find matches, check for occurrences, and manipulate the matched text.

* + - **Creating a Matcher**: You create a Matcher by calling the matcher() method on a Pattern object, passing the input string you want to search.
    - **Matching Methods**: The Matcher class offers several useful methods:
      * find(): Searches for the next occurrence of the pattern in the input string.
      * matches(): Checks if the entire input string matches the pattern.
      * replaceAll(String replacement): Replaces all occurrences of the pattern in the input string with a specified replacement string.
      * group(): Returns the last matched substring.
  + **PatternSyntaxException:** is an exception in Java that occurs when there is an error in the syntax of a regex pattern being compiled..

## **Basic Regex Syntax**

1. ^ Matches beginning of line.
2. $ Matches end of line.
3. . Matches any single character except newline.
4. [...] Matches any single character in brackets.
5. [^...] Matches any single character not in brackets.
6. re\* Matches 0 or more occurrences of preceding expression.
7. re+ Matches 1 or more of the previous thing.
8. re? Matches 0 or 1 occurrence of preceding expression.
9. re{n} Matches exactly n number of occurrences of preceding expression.
10. re{n,} Matches n or more occurrences of preceding expression.
11. re{n,m} Matches at least n and at most m occurrences of preceding expression.
12. (?i) is a special regex flag that makes the pattern case-insensitive.
13. a|b Matches either a or b.
14. ( ) Groups regular expressions and remembers matched text.
15. ?: Groups regular expressions without remembering matched text.
16. ? > re Matches independent pattern without backtracking.
17. \w Matches word characters.
18. \W Matches nonword characters.
19. \s Matches whitespace. Equivalent to [\t\n\r\f].
20. \S Matches nonwhitespace.
21. \d Matches digits. Equivalent to [0-9].
22. \D Matches nondigits.
23. \A Matches beginning of string.
24. \Z Matches end of string. If a newline exists, it matches just before newline.
25. \z Matches end of string.
26. \G Matches point where last match finished.
27. \n Back-reference to capture group number "n".
28. \b Matches word boundaries when outside brackets. Matches backspace 0x08 when inside brackets.
29. \B Matches nonword boundaries.
30. \n, \t, etc. Matches newlines, carriage returns, tabs, etc.
31. \Q Escape quote all characters up to \E.
32. \E Ends quoting begun with \Q.

• Localization and Internationalization

**Chapter 15**

**Day 3**

**Files, Streams and Object serializations**

In this chapter, we explain how

* Java programs create, update and process files.
* we explain that data can be stored in text files and binary files—and we cover the differences between them.
* We demonstrate retrieving information about files and directories using classes Paths and Files and interfaces Path and DirectoryStream (all from package java.nio.file),
* consider the mechanisms for writing data to and reading data from files.
* We show how to create and manipulate sequential-access text files. Working with text files allows you to quickly and easily start manipulating files. As you’ll learn, however, it’s difficult to read data from text files back into object form. Fortunately, many object-oriented languages (including Java) provide ways to write objects to and read objects from files (known as object serialization and deserialization). To demonstrate this, we recreate some of our sequential-access programs that used text files, this time by storing objects in and retrieving objects from binary files.

**What is a file?**

**A file** is a reference to a location on a storage device where data is stored. There are temporary and persistent data storage.

**Temporary Data storage**

* Datastored in variables, arrays and other data structure are temporary. It is lost when a local variable goes out of scope or when the program terminates because both variables and arrays exist in memory (RAM). RAM is a type of volatile memory, meaning that it stores data temporarily while the computer is powered on. Once you turn off your computer or the program that uses that memory ends, all data in RAM is lost.

For example, when you declare a variable like int x = 10; the value 10 is stored in RAM. This storage lasts only for the duration of the program's execution.

For long-term retention of data, even after the computer is turn off or the programs that create the data terminates, computers use persistent data storage.

**Persistent data storage**

This refers to the characteristic of data that outlasts the execution of the program that created it and this includes hard drives, SSDs, flash drives, etc. Data is stored in a persistent medium (like files, databases, or other forms of long-term storage), it remains available even after the program terminates or the computer is turned off.

**Files and Streams**

In Java, files are treated as a sequence of bytes (a byte consists of **8 bits**), meaning that when you read or write to a file, you're working with data in a linear fashion—one byte after another.

**Linear Fashion**:

When we say you're working with data in a linear fashion, it means you typically read from or write to a file one piece of data at a time, sequentially from the beginning to the end of the file. This is similar to reading a book from the first page to the last.

**File Structure**:

Files are structured in a way that the data is stored in a sequence. For example, if you have a text file with the following content:

Hello

World

Java

When you read this file, you start at the first character of "Hello" and move to the next character until you reach the end of the file.

How to identify when you have reached the end of a file when reading from a file. Each operating system has a way to identify when you've reached the end of a file. This could be done through:

* + **EOF Marker**: Some systems use a specific byte or character to indicate that there are no more bytes to read.
  + **Byte Count**: Others keep track of the total number of bytes in a file, so when you reach that number, you know you've reached the end.

**Java File Handling**:

In Java, you often use streams to read or write files.

**What is a Stream?**

A stream in Java is a sequence of data elements that can be read from or written to. It represents a flow of bytes or characters, enabling input/output operations.

**End of Stream**:

When a Java program reads from a stream (like a file), it gets notified by the operating system when it reaches the end of that stream. This means that the program doesn’t need to know how files are stored on disk or how they are represented at the system level. It focuses on processing the data.

**End-of-Stream Indication**

There are two primary ways Java can indicate that it has reached the end of a stream:

* 1. **Exception**: In some cases, trying to read beyond the end of the stream will throw an exception (like EOFException).
  2. **Return Value**: In other cases, a method will return a specific value (often null or -1) when the end of the stream is reached. For instance, read() methods on input streams may return -1 to indicate that there’s no more data to read.

**Byte-Based and Character-Based Streams**

**Byte-Based Streams (0s and 1s).**

Byte-based streams handle data as raw bytes. They are used for binary data input and output. Each data type consumes a specific number of bytes:

* + - A char (character) is typically **2 bytes**.
    - An int (integer) is **4 bytes**.
    - A double (floating-point number) is **8 bytes**.

**File Types**:

* + Files created with byte-based streams are known as **binary files**. These files contain data in a format that is not human-readable, and they can include images, audio, or any type of binary data.
  + Binary files can be processed directly by programs that understand their format. For instance, you can perform calculations with numbers stored in a binary file.

**Character-Based Streams**

Character-based streams handle data as characters, which are represented in a human-readable format.

Each character is typically represented by **2 bytes** (especially in Unicode, which Java uses).

* + The number of bytes required for a value depends on how many characters it contains:
    - The number 2000000000 requires **20 bytes** (10 characters × 2 bytes).
    - The number 7 requires **2 bytes** (1 character × 2 bytes).

**File Types**:

* + Files created with character-based streams are known as **text files**. These files can be easily read and edited using text editors.
  + Text files are great for storing data that needs to be human-readable. For example, a string containing "Sarah Miller is 15 years old" treats the number 15 as a sequence of characters rather than a numeric value that can be calculated directly.

**Standard Streams in Java**

A Java program opens a file by creating an object and associating a stream of bytes or characters with it. The object’s constructor interacts with the operating system to open the file. Java can also associate streams with different devices. When a Java program begins executing, it creates three stream objects that are associated with devices. They are:

1. System.in
2. System.out
3. System.err

**System.in (Standard Input Stream)**

* + **Purpose**: Used for input.
  + **Default Source**: Usually reads input from the keyboard.
  + **Example Use**: When you want to read user input using classes like Scanner

Scanner scanner = new Scanner(System.in);

String userInput = scanner.nextLine(); // Reads a line from the keyboard

1. **System.out (Standard Output Stream)**
   * **Purpose**: Used for output.
   * **Default Destination**: Outputs text to the console (screen).
   * **Example Use**: When you want to print messages or results.

System.out.println("Hello, World!"); // Prints to the console

1. **System.err (Standard Error Stream)**
   * **Purpose**: Used for outputting error messages.
   * **Default Destination**: Also outputs to the console, but typically used to display error messages.
   * **Example Use**: When you want to print error information.

System.err.println("An error occurred!"); // Prints an error message

**Redirection of Streams**

You can change where these streams read from or write to.

* + For **System.in**: you can redirect it to read from a file or another input source by using setIn() of the System class to change the default stream.
  + For **System.out:** you can redirect it to read from a file or another input source by using setOut() of the System class to change the default stream.
  + and **System.err**: you can redirect the output to a file instead of the console by using setErr() methods of the System class to change the default stream

**The java.io and java.nio Packages**

Java programs perform stream-based processing with classes and interfaces from package java.io and the subpackages of java.nio

**Using java.nio package**

The NIO (New Input/Output) package in Java provides a robust set of classes and interfaces for working with files and directories.

**Key Interfaces and Classes**

**Path (Interface)**:

Think of a Path as a way to represent the location of a file or directory on your computer. It does not handle file operations directly; it merely represents the path as an object.

**Example**: If you have a file located at C:/Users/Hp/Desktop/MyFile.txt, you can create a Path object to represent that location.

**Paths (Class)**:

Paths is a utility class that provides static methods for creating Path objects. It acts as a factory to create Path instances based on string representations of file paths.

Example using the **get** method: Paths.get(String first, String... more) creates a Path instance from a string or multiple strings representing the path.

**Files (Class)**:

Provides static methods for various file operations, like:

* + - Copying files: Files.copy(source, target, StandardCopyOption... options)
    - Deleting files: Files.delete(Path path)
    - Checking file existence: Files.exists(Path path)
    - Reading file attributes: Files.readAttributes(Path path, Class<A> type, LinkOption... options)
    - Reading file contents: Files.readAllLines(Path path)

| 1. **Operation** | **Method(s) Used** |
| --- | --- |

|  |  |
| --- | --- |
| Create a file | createNewFile() |

|  |  |
| --- | --- |
| Delete a file | delete() |

|  |  |
| --- | --- |
| Write to a file | FileWriter.write() |

|  |  |
| --- | --- |
| Read from a file | Scanner.nextLine(), BufferedReader.readLine() |

|  |  |
| --- | --- |
| Check if file exists | exists() |

|  |  |
| --- | --- |
| Get file details | getName(), getAbsolutePath(), length(), canRead(), canWrite() |

|  |  |
| --- | --- |
| Rename a file | renameTo() |

|  |  |
| --- | --- |
| Create a directory | mkdir() |

|  |  |
| --- | --- |
| List files in a directory | listFiles() |

|  |  |
| --- | --- |
| Append to a file | FileWriter(true) |

|  |  |
| --- | --- |
| Copy a file | Files.copy() |

|  |  |
| --- | --- |
| Move a file | renameTo() |

|  |  |
| --- | --- |
| Delete a directory | delete() |

**DirectoryStream Interface**:

* **Purpose**: DirectoryStream is used in Java to **read the contents of a directory** (like a folder on your computer) one by one, instead of loading all the files at once.
* **Why Use It?**: When you have a directory with many files, loading all those files into memory can be slow and use a lot of resources. DirectoryStream helps you avoid that by allowing you to iterate through files without loading everything at once.

Sequential-access text files.

Mechanisms for writing data.

Reading data from files:

Challenges of parsing text files back into objects.

**Object Serialization and Deserialization**

In programming, **object serialization** and **object deserialization** are processes used to convert objects into a storable or transmittable format (serialization) and then reassemble them into objects (deserialization). This is often used for saving the state of objects to files or sending them over networks.

**Object Serialization**

**Serialization** is the process of converting an object into a format that can be easily stored or transmitted. Typically, this format is a byte stream that can be written to files, databases, or transmitted over a network.

* In Java, **serialization** can be done using ObjectOutputStream, and the resulting object is stored in a binary file.
* Serialization is used to persist the state of objects so that they can be restored later.

For instance, if you have an object representing a user or a product, serialization allows you to save that object’s state to a file or send it to another machine.

**Key Concept:**

* In **Java**, the class whose objects you want to serialize must implement the Serializable interface. This tells the JVM that the object can be serialized.

**Object Deserialization**

**Deserialization** is the reverse process of serialization. It involves reading a previously serialized byte stream and converting it back into an object of the original class.

* The byte stream is read using ObjectInputStream, and the data is converted back to an object.
* Deserialization allows you to read the data from a storage medium, like a file, and reinstantiate the object with its original values.

**Advantages of Using Binary Files for Storing Objects**

Using binary files for serialization has several advantages, especially when dealing with complex objects:

* **Compact storage**: Binary files tend to take up less space than text-based formats (like JSON or XML).
* **Faster read/write**: Reading and writing binary files is faster because the data is in a compact binary format rather than human-readable text.
* **Preserving object structure**: Binary serialization preserves the exact structure of an object, including its private fields, unlike some text-based formats that may require extra parsing.
* **Security**: Binary files are not as easily readable by humans, which can provide a level of obfuscation, though this does not mean they are secure from unauthorized access.

However, one of the downsides of binary serialization is that the format is not human-readable, and any changes to class definitions might break compatibility with serialized data (a versioning problem).

**Example: Recreating Sequential-Access Programs Using Binary Files**

In sequential-access programs, data is read or written in a specific order, often one record at a time. This can be useful in scenarios where you need to store objects and retrieve them in a specific sequence.

Day 4

**Week 8**

**Chapter 16**

**Generic collections**

A **generic collection** in Java is simply a **container** that stores **objects**. However, what makes it "generic" is that it allows you to specify the **type** of objects that the collection will hold.

This helps to **prevent errors** at compile time and allows you to work with collections in a more **flexible and type-safe** way.

Think of it like a box:

* If you have a **generic box**, you can specify what type of things (e.g., books, toys, or shoes) it will hold.
* If you don't specify the type, the box will be able to hold anything (but you won't know for sure what’s inside).
* When you specify the type of items the box can hold (e.g., a **box for books**), the box will only allow books inside and **not any other items**.

**Types of collection in java**

Java provides several types of collections, all part of the java.util package.

**1. List**

A List is an ordered collection that allows duplicate elements. It maintains the order of insertion, so the elements can be accessed by their index.

**Examples of list are:**

* **ArrayList**: A dynamic array that grows as elements are added. It's fast for random access but slower for insertions/deletions in the middle of the list.

**When to Use**:

Use ArrayList when you need fast access to elements using indices, and don’t need to do frequent insertions or deletions in the middle or at the beginning.

* **LinkedList**: A doubly linked list, where elements are linked with each other. It’s slower for random access but efficient for inserting and removing elements at the beginning or middle.

**When to Use**:

Use LinkedList when you frequently need to insert or remove elements at the beginning or middle of the list, and you don't need to access elements by index frequently.

**2. Set**

A Set is an unordered collection that does not allow duplicate elements. It's useful when you want to store a group of unique items.

* **HashSet**: Implements the Set interface, backed by a hash table. It doesn't guarantee any specific order.
* **LinkedHashSet**: Similar to HashSet, but it maintains the insertion order of elements.
* **TreeSet**: Implements a Set that is ordered according to the natural ordering of its elements or by a comparator provided at set creation.

**3. Queue**

A Queue represents a collection designed for holding elements prior to processing. It is often used in scenarios like task scheduling or message processing.

* **PriorityQueue**: A queue where elements are ordered based on their priority. It doesn't necessarily follow the order of insertion.
* **LinkedList (Queue implementation)**: Can also be used as a queue, as it implements the Queue interface.

**4. Map**

A Map is a collection that stores key-value pairs. It allows you to map a unique key to a corresponding value.

* **HashMap**: A basic implementation of the Map interface, where keys and values are stored in a hash table. It doesn’t guarantee any order.
* **LinkedHashMap**: Similar to HashMap, but maintains the insertion order of the keys.
* **TreeMap**: Implements the Map interface and orders the keys according to their natural ordering or a custom comparator.
* **Hashtable**: A legacy collection, similar to HashMap but synchronized.

**5. Special Collections**

These are collections designed for specific use cases.

* **Stack**: A collection that follows the Last-In-First-Out (LIFO) principle. The Stack class is part of Java’s standard library but is less commonly used today in favor of Deque.
* **Deque**: A double-ended queue that allows adding/removing elements from both ends. The ArrayDeque class is often used here.

**6. Synchronized Collections**

These are special types of collections that ensure thread-safety, i.e., they can be safely used in multi-threaded environments.

* **SynchronizedList, SynchronizedSet, SynchronizedMap**: These are synchronized versions of List, Set, and Map from the Collections utility class.

**7. Unmodifiable Collections**

These are collections that cannot be modified once created. They can be useful for situations where you want to protect the integrity of data.

* **UnmodifiableList, UnmodifiableSet, UnmodifiableMap**: These are unmodifiable views of the original collections provided by the Collections utility class.

**Summary of Major Collections:**

* **List**: Ordered, allows duplicates (e.g., ArrayList, LinkedList).
* **Set**: Unordered, no duplicates (e.g., HashSet, TreeSet).
* **Queue**: For processing elements in order (e.g., PriorityQueue, LinkedList).
* **Map**: Stores key-value pairs (e.g., HashMap, TreeMap).
* **Deque**: Double-ended queue (e.g., ArrayDeque).
* **Stack**: LIFO order (e.g., Stack class, though less commonly used now).

Day 1

• List

• Iterator

• Set

Day 2

• Deque

• Map

• Class Collections methods

• Comparator interface

\*\*Chapter 17, 18, 19

Chapter 20

Day 3

• Generic methods and classes

• Wildcards in generics

Chapter 23

Day 4

•\*\* Concurrency

• Thread Synchronization

Week 9

Chapter 23

Day 1

• Lock and Conditions

Chapter 24

**Accessing Databases With JDBC**

Day 2

**Database**

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation.

A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

1. **SQL (Structured Query Language)**: This is the language used to communicate with relational databases. SQL is used to perform two primary tasks:
   * **Queries**: Requesting information based on certain criteria.
   * **Data Manipulation**: Modifying, adding, or deleting data in the database.
2. **Pronunciation**: There's a common debate on how to pronounce SQL. Some people pronounce it "sequel" (like the word for a follow-up or continuation of something), while others pronounce it by saying the individual letters: "S-Q-L." The passage mentions that in this context, the pronunciation "sequel" is preferred.
3. **International Standard**: SQL is recognized as the international standard for relational databases, making it widely adopted across many different database systems (such as MySQL, PostgreSQL, SQL Server, and Oracle).

**Popular Relational Database Management Systems**

Some popular relational database management systems (RDBMSs) are Microsoft SQL Server®, Oracle®, Sybase®, IBM DB2®, Informix®, PostgreSQL and MySQL™. The JDK comes with a pure-Java RDBMS called Java DB—the Oracle-branded version of Apache Derby™.

### JDBC (Java Database Connectivity)

JDBC Java programs interact with databases using the Java Database Connectivity (JDBC™) API. A JDBC driver enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

**Relational Database**

A relational database organizes and stores data in a way that allows for efficient access and manipulation without the user needing to worry about the physical structure of the data. The core concept behind a relational database is the use of tables, which consist of rows and columns.

1. **Tables**:
   * A table in a relational database consists of a set of rows (also known as records or tuples) and columns (also known as attributes or fields).
   * Each row in the table represents an entity or record, and each column represents an attribute or property of that entity.
2. **Primary Key**:
   * Every table in a relational database has a **primary key**—a unique identifier for each row. This ensures that no two rows in the table can have the same value for the primary key.
   * The primary key could be a single column or a combination of columns, but it must always guarantee uniqueness across rows. Examples include Social Security Numbers, employee ID numbers, or part numbers in an inventory system.
3. **Relational Structure**:
   * The **relational** part of relational databases refers to how data is related between different tables. Relationships are established using **foreign keys**, which are references to primary keys in other tables. These relationships can be used to join data across multiple tables, allowing for more complex queries and data management.

### Example:

In the personnel system described in your example, the table could look something like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number (Primary Key)** | **Name** | **Department** | **Hire Date** | **Salary** |
| 1001 | Alice Smith | IT | 2022-01-15 | 75000 |
| 1002 | Bob Johnson | HR | 2020-04-11 | 68000 |
| 1003 | Charlie Brown | IT | 2021-08-05 | 72000 |
| 1004 | Dana White | Finance | 2023-03-20 | 80000 |
| 1005 | Eve Davis | HR | 2019-11-25 | 65000 |
| 1006 | Frank Black | IT | 2022-06-17 | 74000 |

* The **Number** column serves as the **primary key** for this table. Each row (representing an employee) can be uniquely identified by their employee number.
* The **columns** (Name, Department, Hire Date, Salary) represent attributes of the employee.

### Order of Rows:

While the rows in the table are displayed in ascending order by primary key in this example, the order in which rows appear in a table is not important unless specifically queried with an **ORDER BY** clause in SQL. The relational model ensures that the rows can be accessed or manipulated based on the data, not their physical arrangement.

### Additional Considerations:

* **Foreign Keys**: These are columns in a table that reference the primary key of another table. For example, in an employee table, you could have a foreign key linking each employee to a specific department in a separate "Departments" table.
* **Normalization**: Relational databases often employ **normalization** to minimize redundancy and ensure that the database structure is efficient and consistent.

**A books Database**

In the **books database** described in the text, we are dealing with three tables: **Authors**, **Titles**, and **AuthorISBN**. These tables work together to store information about authors, their books, and the many-to-many relationships between them. Let's break down the structure of each table and the relationships between them, along with an explanation of key concepts such as primary keys, foreign keys, and relationships.

### 1. ****Authors Table****

The **Authors** table stores information about the authors, including a unique ID number, first name, and last name.

|  |
| --- |
| **Structure of the Authors Table**: |
| |  |  | | --- | --- | | **Column Name** | **Description** | | **AuthorID** | The unique identifier for each author (Primary Key). This column is defined as **autoincremented**, meaning each new row gets a unique value automatically. | | **FirstName** | The author's first name. | | **LastName** | The author's last name. | |

#### Example Data:

|  |  |  |
| --- | --- | --- |
| **AuthorID** | **FirstName** | **LastName** |
| 1 | Paul | Deitel |
| 2 | Harvey | Deitel |
| 3 | Abbey | Deitel |
| 4 | Dan | Quirk |
| 5 | Michael | Morgano |

* **Primary Key**: AuthorID is the primary key, ensuring each author has a unique identifier.

### 2. ****Titles Table****

The **Titles** table stores information about books, including their **ISBN** (International Standard Book Number), title, edition number, and copyright year.

#### **Structure of the Titles Table:**

|  |  |
| --- | --- |
| **Column Name** | **Description** |
| **ISBN** | The unique identifier for each book (Primary Key). |
| **Title** | The title of the book. |
| **EditionNumber** | The edition number of the book. |
| **Copyright** | The copyright year of the book. |

#### **Example Data:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ISBN** | **Title** | **EditionNumber** | **Copyright** |
| 0132151006 | Internet & World Wide Web How to Program | 5 | 2012 |
| 0133807800 | Java How to Program | 10 | 2015 |
| 0132575655 | Java How to Program, Late Objects Version | 10 | 2015 |
| 013299044X | C How to Program | 7 | 2013 |
| 0132990601 | Simply Visual Basic 2010 | 4 | 2013 |

* **Primary Key**: ISBN is the primary key for the Titles table, ensuring that each book has a unique ISBN.

### 3. ****AuthorISBN Table****

The **AuthorISBN** table links authors and books together. It contains two columns:

* **AuthorID** (foreign key referencing AuthorID in the **Authors** table).
* **ISBN** (foreign key referencing ISBN in the **Titles** table).

This table represents the many-to-many relationship between authors and books. Each author can write multiple books, and each book can have multiple authors. The **AuthorISBN** table creates a connection between these two entities.

#### **Structure of the AuthorISBN Table:**

|  |  |
| --- | --- |
| **Column Name** | **Description** |
| **AuthorID** | Foreign key referencing the AuthorID in the **Authors** table. |
| **ISBN** | Foreign key referencing the ISBN in the **Titles** table. |

#### **Example Data:**

|  |  |
| --- | --- |
| **AuthorID** | **ISBN** |
| 1 | 0132151006 |
| 2 | 0132151006 |
| 1 | 0133807800 |
| 4 | 0132151006 |
| 2 | 0132575655 |
| 5 | 013299044X |
| 3 | 0132990601 |

* **Composite Primary Key**: The combination of AuthorID and ISBN forms a **composite primary key**. This means each row in this table is uniquely identified by the pair of values (AuthorID, ISBN).
* **Foreign Keys**:
  + AuthorID is a foreign key referencing the **Authors** table.
  + ISBN is a foreign key referencing the **Titles** table.

### Relationships Between Tables

The **Authors**, **Titles**, and **AuthorISBN** tables are related as follows:

1. **One-to-Many Relationship Between Authors and AuthorISBN**:
   * One author can write multiple books, but each row in the **AuthorISBN** table links one author to one book. For example, **AuthorID 1 (Paul Deitel)** is associated with multiple ISBNs in the **AuthorISBN** table, meaning Paul has written multiple books.
2. **One-to-Many Relationship Between Titles and AuthorISBN**:
   * One book (identified by its **ISBN**) can have multiple authors. For example, **ISBN 0132151006** appears multiple times in the **AuthorISBN** table, meaning multiple authors have contributed to that book.
3. **Many-to-Many Relationship Between Authors and Titles**:
   * An author can write many books, and a book can have many authors. This many-to-many relationship is established via the **AuthorISBN** table. This is crucial for accurately modeling the situation where authors collaborate on books, or books are revised and updated by multiple authors.

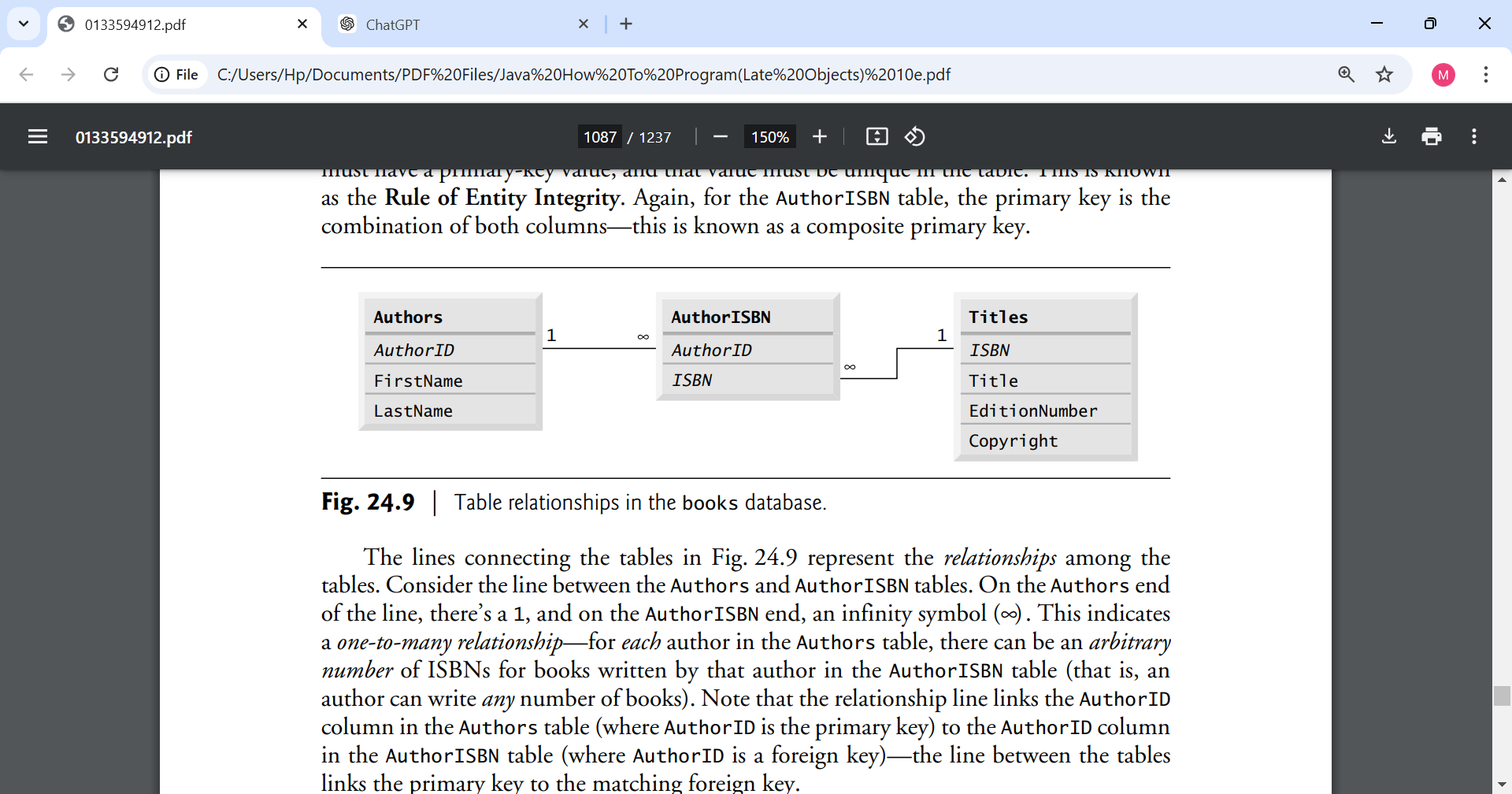
### Entity-Relationship (ER) Diagram

The **ER Diagram** provides a visual representation of the relationships among the tables:

* The **Authors** table has a one-to-many relationship with the **AuthorISBN** table (one author can be linked to many books).
* The **Titles** table also has a one-to-many relationship with the **AuthorISBN** table (one book can be linked to many authors).

The **AuthorISBN** table forms the many-to-many relationship between **Authors** and **Titles**.

**Entity-Relationship Diagram**



### Referential Integrity

* **Rule of Referential Integrity**: Foreign key values must match primary key values in the referenced table. For example, the **AuthorID** in the **AuthorISBN** table must exist as a valid **AuthorID** in the **Authors** table.
* This ensures that no "orphan" records exist, meaning every row in the **AuthorISBN** table refers to a valid author and a valid book.

### Conclusion

This books database is a good example of how relational databases use **primary keys**, **foreign keys**, and **composite keys** to represent relationships between entities. The **Authors** and **Titles** tables are linked through the **AuthorISBN** table, which resolves the many-to-many relationship between authors and books. By using these relational concepts, the database is able to maintain data integrity and provide a clear, structured way to store and retrieve information about authors, their books, and their collaborations.

1. **DriverManager**: This class manages a list of database drivers. When a connection request is made, it is responsible for choosing the appropriate driver based on the database URL.
2. **Connection**: The Connection object represents an open connection to the database. It is used to create Statement, PreparedStatement, and CallableStatement objects to execute SQL queries and commands.
   * **Methods**: getConnection(), createStatement(), prepareStatement(), etc.
   * **Connection URL**: Used to identify the location and the type of database.
3. **Statement**: The Statement object is used to execute SQL queries against the database. There are three types of statements:
   * **Statement**: Used for simple SQL queries (non-parameterized).
   * **PreparedStatement**: Used for SQL queries that can accept parameters. It is more efficient and secure than a Statement because it prevents SQL injection.
   * **CallableStatement**: Used to execute stored procedures in the database.
4. **ResultSet**: The ResultSet object represents the result set of a query. It is used to iterate through and extract data from the database.
   * **Methods**: next(), getString(), getInt(), close(), etc.
   * **Types of ResultSets**: Forward-only, scrollable, updatable.
5. **SQLException**: A class that handles exceptions thrown during JDBC operations. It provides detailed information about the error, such as error codes and descriptions.
6. **DataSource**: An alternative to DriverManager that provides better management and configuration of database connections, especially in enterprise applications. It supports connection pooling.

### Steps for Using JDBC

Here’s a step-by-step guide to how JDBC works:

1. **Load the Database Driver**:  
   The first step is to load the appropriate JDBC driver for the database you are using (though newer JDBC versions support automatic driver loading):

java

Copy code

Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL

Or, with JDBC 4.0 and later, this is usually not needed since drivers are automatically loaded from the classpath.

1. **Establish a Database Connection**:  
   Use the DriverManager or DataSource to establish a connection to the database:

java

Copy code

Connection conn = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/mydatabase", "username", "password");

1. **Create a Statement Object**:  
   Once the connection is established, a Statement or PreparedStatement object is created to execute SQL queries:

java

Copy code

Statement stmt = conn.createStatement();

1. **Execute SQL Queries**:  
   You can use the executeQuery() method to execute SELECT queries, and executeUpdate() for INSERT, UPDATE, DELETE, and other non-SELECT operations:

java

Copy code

ResultSet rs = stmt.executeQuery("SELECT \* FROM users");

1. **Process the ResultSet**:  
   If the SQL query returns data (SELECT statement), the ResultSet is processed:

java

Copy code

while (rs.next()) {

String username = rs.getString("username");

System.out.println(username);

}

1. **Close the Resources**:  
   After the database operations are complete, it is important to close the ResultSet, Statement, and Connection to release database resources:

java

Copy code

rs.close();

stmt.close();

conn.close();

• JDBC • SELECT, INSERT, UPDATE, DELETE

Day 3

• SELECT, INSERT, UPDATE, DELETE continued

• ResultSetMetaData

Week 10

Class case study