**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**

**1. Web Development:** Java is extensively used for building robust, scalable server-side applications. Frameworks like **Spring**, **Spring Boot**, and **Java EE** (now Jakarta EE) are commonly used for creating enterprise-level web applications and RESTful APIs.

**2. Mobile Development:** Although Kotlin is now the preferred language for Android development, Java remains a popular choice for writing Android apps. The Android SDK is built on Java, and many legacy Android apps are written in Java.

**3. Enterprise Applications:** Java is the go-to language for building large-scale, complex enterprise applications. The **Java Enterprise Edition** (Jakarta EE) provides tools for building distributed, multi-tiered, and transactional applications for businesses, including customer relationship management (CRM) and enterprise resource planning (ERP) systems.

**4. Desktop Applications:** Java can be used to create cross-platform desktop applications using technologies like **Swing** and **JavaFX**. While not as commonly used for desktop development today as other languages, it remains a solid choice for building graphical user interface (GUI) applications.

**5. Big Data Technologies:** Java is often used in big data ecosystems. Tools like **Apache Hadoop**, **Apache Spark**, and **Apache Kafka** have Java APIs, and many big data processing tasks are written in Java.

**6. Scientific and Research Applications:** Java is used in scientific computing, natural language processing, and other research applications. Its portability and scalability make it a good fit for high-performance computing tasks.

**7. Embedded Systems:** Java can be used in embedded systems and devices like smart cards, sensors, and IoT (Internet of Things) devices, with platforms such as **Java ME** (Micro Edition).

**8. Cloud Computing:** Many cloud platforms and services, including those from **Amazon Web Services (AWS)**, **Google Cloud**, and **Microsoft Azure**, provide SDKs and tools for developing cloud-based applications in Java. It's widely used in cloud-based backend services, microservices, and serverless computing.

**9. Gaming:** While not as popular as C++ or Unity for game development, Java has been used in the creation of some games, particularly through platforms like **LibGDX** and **Minecraft**, which was originally written in Java.

**10. Financial Services:** The finance industry relies heavily on Java for building high-performance, low-latency trading systems, banking software, and financial management tools due to its reliability and scalability.

**11. Scripting and Automation:** Java is also used for creating automated tools, schedulers, and scripts for managing system processes, data flows, and enterprise workflows.

**12. Machine Learning & AI:** While Python dominates machine learning, Java has libraries like **Weka**, **Deeplearning4j**, and **MOA** (Massive Online Analysis) that are used for data analysis, machine learning, and AI development.

**13. Internet of Things (IoT):** Java is commonly used in IoT systems due to its scalability and cross-platform support. Java ME Embedded and other Java frameworks are specifically designed for IoT environments.

**Java Terminology**

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

1. **Java Development Kit (JDK):**The Java Development Kit (JDK) is a software development kit provided by Oracle that includes everything you need to develop Java applications. It contains several key components:

* **Java Compiler (javac):** This tool translates Java source code (.java files) into bytecode (.class files) that the Java Virtual Machine (JVM) can execute.
* **Java Runtime Environment (JRE**): In simple terms, the Java Runtime Environment (JRE) is like the operating system for Java programs. It provides everything needed to run a Java application on your computer. It includes the following
  + **Java Virtual Machine (JVM)**: Think of the Java Virtual Machine (JVM) as a translator and manager for Java programs. When you write a Java program, it gets turned into a special kind of code called bytecode, which is not directly understood by your computer’s hardware. The JVM steps in to interpret this bytecode and run the program on your computer. It ensures that the program works the same way no matter what type of computer or operating system you’re using. So, in simple terms, the JVM makes sure Java programs can run on any device without needing to be rewritten. This is why Java is known as a**platform-independent language.**
* **Libraries and Files**: The JRE includes various libraries (pre-written code) and files that Java programs need to function correctly. These libraries help Java applications perform common tasks, like reading files or handling user input.
* **No Development Tools**: The JRE doesn’t include tools for writing or compiling Java code. It only provides the environment necessary to run Java applications that have already been written and compiled.
* **Development Tools:** The JDK includes various tools for debugging, monitoring, and managing Java applications. Some examples are:
* **javap:** For disassembling and viewing the bytecode of Java classes.
* **javadoc:** For generating documentation from Java source code.
* **jdb:** For debugging Java programs.

Other Utilities: It also comes with other utilities and libraries that assist in developing Java applications.

In short, if you want to develop Java software, you need the JDK. It provides all the tools and components necessary for writing, compiling, and running Java code.

**There are three execution phases of a program. They are**

1. **Writing phase:** The writing phase involves writing the program which is done by a java programmer like you and me.
2. **Compilation phase:** The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
3. **Running phase:** In the Runningphase of a program,**JVM** executes the bytecode generated by the compiler.

**2. Garbage Collector:**Think of garbage collection like cleaning up a messy room. When you use a room (or in this case, a computer's memory), you accumulate stuff (like old papers or broken items) that you no longer need. Over time, this stuff takes up space and can make things cluttered and harder to manage.

In computing, **garbage collection** is the process that automatically cleans up unused memory in your computer:

* **Memory Allocation**: When a program runs, it uses memory to store data and objects. Over time, some of this data and objects are no longer needed.
* **Automatic Cleanup**: The garbage collector is like a cleaning service that periodically scans the memory to find and remove these unused objects. This helps free up memory space so that new data can be stored efficiently.
* **No Manual Effort**: You don’t need to manually manage this cleanup process. The garbage collector handles it automatically in the background, which help to keeps your program running smoothly. It is the work of JVM to inform the garbage collector to automatically cleanup.

In essence, garbage collection is the system's way of keeping the memory tidy, so your programs can run efficiently without you having to worry about cleaning up after them.

**Main Features of Java**

**1. Platform Independent:**Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the bytecode. That is why we call java a platform-independent language.

**2. Object-Oriented Programming:**Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The four main concepts of Object-Oriented programming are:

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**3.** **Simple:**Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.

**4.** **Robust:**Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

**5.** **Safe and Secure**: Java has built-in safety features that help prevent common programming mistakes, like accessing invalid memory. It's like having a safety net that catches errors before they cause problems.

**6.** **Distributed:**We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

**7.** **Multithreading:**Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

**8.** **Portable:**As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

**9. High Performance:** Java architecture is defined in such a way that it reduces overhead during the runtime and at sometimes java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.

**10. Dynamic flexibility:**Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

**12. Write Once Run Anywhere:** As discussed above java application generates a ‘.class’ file that corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

**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)****: ****Java SE**** is the core version of the Java programming language and platform. It provides the foundational libraries and APIs required for developing desktop, server, and general-purpose applications. Java SE contains the ****JVM****, ****JRE****, and essential libraries that are used for programming in Java.

**Key Features**:

* + **Core APIs**: Includes essential libraries like **java.lang**, **java.util**, **java.io**, **java.net**, etc.
  + **Multithreading**: Supports **multi-threading** to develop applications that can handle multiple tasks at the same time.
  + **Networking**: Java SE includes classes to work with networking protocols such as **TCP/IP**, HTTP, etc.
  + **JavaFX (UI)**: Although JavaFX is a separate framework, Java SE can be used to develop **graphical user interface (GUI)** applications with JavaFX.
  + **Java Collections Framework**: Provides powerful data structures like ArrayList, HashMap, etc.
  + **Functional Programming (Java SE 8+)**: Introduced in Java SE 8, **Lambda Expressions**, **Streams API**, and **Functional Interfaces** enable functional-style programming within Java.

**Primary Use Cases**:

* + **Desktop applications** (e.g., GUI applications using JavaFX).
  + **Server-side applications** (e.g., APIs, services).
  + **Console-based applications**.

### 2. ****Java Enterprise Edition (Java EE)****: **Java EE** (now **Jakarta EE** since 2018) is an extended version of Java SE that provides a set of specifications for developing **enterprise-grade applications**. These applications are typically **large-scale**, **distributed**, and **web-based**. Java EE includes APIs and tools for creating **web applications**, **enterprise services**, **message-driven services**, and **database integration**.

**Key Features**:

* + **Web and Enterprise Services**: Includes technologies like **Servlets**, **JavaServer Pages (JSP)**, **EJB (Enterprise JavaBeans)**, and **JAX-RS** (for RESTful web services).
  + **Persistence**: Uses **JPA (Java Persistence API)** for database interaction.
  + **Messaging**: Provides **JMS (Java Message Service)** for message-based communication.
  + **Transactions**: Built-in support for **JTA (Java Transaction API)**, allowing distributed transactions across multiple resources.
  + **Security**: Java EE includes comprehensive security frameworks for authentication, authorization, and secure communication.
  + **WebSocket** and **Java EE Web Services**: Supports real-time communication and building RESTful and SOAP-based web services.

**Primary Use Cases**:

* + **Large-scale, distributed applications** (e.g., banking systems, CRM, ERP).
  + **Web applications** (using technologies like **Servlets**, **JSP**, **JSF**, **RESTful services**).
  + **Enterprise solutions** involving messaging, persistence, and transaction management.
* **Note**: Since Java EE became **Jakarta EE**, its development has shifted under the **Eclipse Foundation**, but its core concepts and technologies remain the same.

### 3. ****Java Micro Edition (Java ME)****: **Java ME** is a subset of Java SE designed for developing applications on **resource-constrained** devices. These devices typically have limited memory, processing power, and storage, such as **mobile phones**, **embedded systems**, **smartwatches**, **MP3 players**, **smart meters**, and more.

**Key Features**:

* + **Lightweight Runtime**: Java ME includes a minimal **runtime environment** (known as the **CLDC** - Connected Limited Device Configuration) that is optimized for small devices.
  + **MIDP (Mobile Information Device Profile)**: This profile provides a standard set of APIs for developing mobile and embedded applications. It supports user interfaces, networking, file I/O, and more for devices with limited capabilities.
  + **Embedded and IoT Devices**: Ideal for **Internet of Things (IoT)** devices and **embedded systems** such as sensors, printers, and home appliances.
  + **Optional Features**: Depending on the device, Java ME offers different configurations and profiles, such as the **CDC (Connected Device Configuration)** for more powerful devices.

**Primary Use Cases**:

* + **Mobile applications** (especially older mobile phones with Java support).
  + **Embedded systems** (e.g., **smartwatches**, **medical devices**, **home appliances**, **automobile systems**).
  + **IoT (Internet of Things)** applications where small devices need to run Java-based software.

### 4. ****JavaFX****: **JavaFX** is a framework for building **rich, modern graphical user interfaces (GUIs)** in Java. It was introduced by Sun Microsystems (now Oracle) as a successor to **Swing**, the older Java GUI framework. JavaFX allows developers to create highly interactive, media-rich applications that can run on **desktop** and **mobile** platforms.

**Key Features**:

* + **Scene Graph**: JavaFX uses a **scene graph** to represent the UI hierarchy, making it easy to work with visual elements like buttons, text, images, and shapes.
  + **FXML**: JavaFX includes **FXML**, a markup language similar to **HTML** or **XML**, which allows for the design of UI components in a declarative way, separating logic and design.
  + **CSS Styling**: JavaFX allows developers to style user interfaces using **CSS** (Cascading Style Sheets), providing flexibility in design.
  + **3D Graphics**: JavaFX has built-in support for 3D graphics, enabling the creation of sophisticated animations and visual effects.
  + **JavaFX Controls**: Offers pre-built **UI components** like buttons, labels, text fields, tables, and charts.
  + **Multimedia**: JavaFX includes built-in support for audio, video, and animations, allowing developers to create media-rich applications.
  + **Cross-platform**: JavaFX applications can run on multiple platforms, including **Windows**, **macOS**, **Linux**, and even **mobile** and **web** (using JavaFX WebView).

**Primary Use Cases**:

* + **Desktop GUI applications** (e.g., media players, enterprise software with rich UIs).
  + **Data visualization** (with JavaFX charts and 3D graphics).
  + **Interactive, multimedia-rich applications**.
  + **Cross-platform applications** (JavaFX applications can run on Windows, macOS, and Linux with minimal changes).
* **Evolution**:
  + **JavaFX** is part of the **Java SE** distribution but can also be used separately. It was decoupled from the standard JDK distribution after JDK 11, and now developers can include it via a separate **open-source** package known as **OpenJFX**.

**Note:** All types of Java programming contain an API (Application Programming Interface) and [JVM (Java Virtual Machine)](https://www.tutorialsfreak.com/java-tutorial/java-jvm). Java API is used to build other software programs, while java JVM is used to run Java programs on a specific platform.

The JVM and API provided by each platform enable the program to run on any compatible system and leverage the benefits of Java programming.

**Data Hierarchy**

Data items processed by computers form a data hierarchy that becomes larger and more complex in structure as we progress from the simplest data items (called “bits”) to richer ones, such as characters and fields.

**Bits**

A bit is the smallest piece of information in a computer, and it’s like a tiny switch that can either be off (0) or on (1). Just as you might flip a switch to turn a light on or off, a bit can be in one of two states. When you combine lots of these switches (bits), you can represent more complex information, like numbers, letters, or pictures. For example, 8 bits together form a "byte," which can represent a single character, like a letter or number.

**Characters**

It’s tedious for people to work with data in the low-level form of bits instead characters are used in representing and processing written information. characters are symbols used in text and data, such as letters, numbers, punctuation marks, and special symbols.

**How characters are handled in computer:**

1. **Encoding:** Characters are represented in computers using encoding standards. Each character is assigned a unique number in an encoding system, which the computer uses to store and manipulate text. Common encoding systems include:
   * **ASCII (American Standard Code for Information Interchange):** Encodes 128 characters, including English letters, digits, and some special symbols. Each character is represented by a 7-bit number.
   * **Unicode:** A more comprehensive system that includes characters from many languages and scripts. It uses different encoding forms like UTF-8**(8-bit Unicode Transformation Format),** UTF-16**(16-bit Unicode Transformation Format)**, and UTF-32**(32-bit Unicode Transformation Format)** to represent characters. Unicode can represent over a million characters.
2. **Examples of Characters:**
   * **Letters:** A, B, C, a, b, c
   * **Digits:** 0, 1, 2, 3
   * **Punctuation Marks:** !, ?, ., , ", :,
   * **Special Symbols:** @, #, $, &, ©, ™
3. **Text Representation:** When you type text into a computer, each character you type is converted into its encoded number, which is stored as binary data. For instance, typing the letter "A" might be stored as a specific binary number in the computer’s memory.
4. **Manipulation:** Characters are used to form strings of text. Computers use various operations to manipulate these strings, such as searching, replacing, and formatting.

**Fields**

Just as characters are composed of bits, fields are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters can be used to represent a person’s name, and a field consisting of decimal digits could represent a person’s age.

**Records**

Several related fields can be used to compose a record (implemented as a class in Java). In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):

* Employee identification number (a whole number)
* Name (a string of characters)
* Address (a string of characters)
* Hourly pay rate (a number with a decimal point)
* Year-to-date earnings (a number with a decimal point)
* Amount of taxes withheld (a number with a decimal point)

**Files**

A file is a group of related records.

**Database**

A database is a collection of data organized for easy access and manipulation. The most popular model is the relational database, in which data is stored in simple tables. A table includes records and fields. For example, a table of students might include first name, last name, major, year, student ID number and grade point average fields. The data for each student is a record, and the individual pieces of information in each record are the fields. You can search, sort and otherwise manipulate the data based on its relationship to multiple tables or databases. For example, a university might use data from the student database in combination with data from databases of courses, on-campus housing, meal plans, etc.

**Big Data**

**big data** refers to an extremely large and complex set of data that is so vast it’s challenging to process and analyze using traditional methods. Imagine trying to analyze the information from every tweet, Facebook post, online purchase, and sensor in the world all at once—that’s what big data is like.

Here are some simple ways to understand big data:

1. **Huge Amounts of Information:** It’s like having a gigantic pile of documents, photos, videos, and other data that’s too large to fit into your standard filing cabinet. Think of the entire internet, all social media posts, or all the data from online transactions.
2. **Speed:** Big data comes in quickly, like a fast-moving river of information that’s constantly flowing. It’s not just about the amount but also about how fast new data is created and needs to be processed.
3. **Variety:** It includes many different types of data—text, numbers, images, videos, and more. Just like having a mix of different kinds of documents, videos, and emails all in one place.

**Machine Language, Assembly Languages and High-level Languages**

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of such languages are in use today.

These may be divided into three general types:

1. Machine languages

2. Assembly languages

3. High-level languages

**Machine Languages:**

Any computer can directly understand only its own machine language, defined by its hardware design. Machine languages generally consist of 1s and 0s. it instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans. For example, here’s a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:

+1300042774

+1400593419

+1200274027

**Assembly Languages and Assemblers**

Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of assembly languages. **Translator programs called assemblers** were developed to convert early assembly-language programs to machine language at computer speeds.

The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay:

load basepay

add overpay

store grosspay

Although such code is clearer to humans, it’s incomprehensible to computers until translated to machine language.

**High-Level Languages and Compilers**

With the advent of assembly languages, computer usage increased rapidly, but programmers still had to use numerous instructions to accomplish even the simplest tasks. To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks. **Translator programs called compilers** convert high-level language programs into machine language. High-level languages allow you to write instructions that look almost like every day English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a single statement such as

grossPay = basePay + overTimePay

From the programmer’s standpoint, high-level languages are preferable to machine and assembly languages. Java is one of the most widely used high-level programming languages. Interpreters Compiling a large high-level language program into machine language can take considerable computer time. Interpreter programs, developed to execute high-level language programs directly, avoid the delay of compilation, although they run slower than compiled programs.

**Keyword in java**

In Java, a **keyword** is a reserved word that has a predefined meaning in the Java language syntax. Keywords cannot be used for anything other than their intended purpose. They are essential to defining the structure and behavior of Java programs.

**List of Java Keywords**

As of Java 21, there are **57** keywords in Java. Here’s a list of all Java keywords:

1. **abstract** - Used to declare an abstract class or method.
2. **assert** - Used for debugging purposes to make an assertion.
3. **boolean** - Used to declare a variable of type boolean (true/false).
4. **break** - Exits from a loop or switch statement.
5. **byte** - Used to declare a variable of type byte (8-bit integer).
6. **case** - Defines a branch in a switch statement.
7. **catch** - Used to handle exceptions in a try-catch block.
8. **char** - Used to declare a variable of type char (single 16-bit Unicode character).
9. **class** - Used to declare a class.
10. **continue** - Skips the current iteration of a loop and proceeds with the next iteration.
11. **default** - Specifies the default case in a switch statement.
12. **do** - Starts a do-while loop.
13. **double** - Used to declare a variable of type double (double-precision floating-point number).
14. **enum** - Defines a set of named constants.
15. **extends** - Indicates that a class is inheriting from a superclass.
16. **final** - Used to define constants, prevent method overriding, or inheritance.
17. **finally** - Used to execute a block of code after a try-catch block, regardless of whether an exception was thrown or not.
18. **float** - Used to declare a variable of type float (single-precision floating-point number).
19. **for** - Starts a for loop.
20. **if** - Starts an if statement.
21. **implements** - Indicates that a class implements an interface.
22. **import** - Imports other classes or entire packages.
23. **instanceof** - Tests whether an object is an instance of a specific class or interface.
24. **int** - Used to declare a variable of type int (32-bit integer).
25. **interface** - Used to declare an interface.
26. **long** - Used to declare a variable of type long (64-bit integer).
27. **native** - Specifies that a method is implemented in native code using JNI (Java Native Interface).
28. **new** - Creates new objects.
29. **null** - Represents a null reference.
30. **package** - Defines a namespace for organizing classes and interfaces.
31. **private** - Specifies that a member is accessible only within its own class.
32. **protected** - Specifies that a member is accessible within its own package and by subclasses.
33. **public** - Specifies that a member is accessible from any other class.
34. **return** - Exits from a method and optionally returns a value.
35. **short** - Used to declare a variable of type short (16-bit integer).
36. **static** - Indicates that a member belongs to the class, rather than instances of the class.
37. **strictfp** - Used to restrict floating-point calculations to ensure portability.
38. **super** - Refers to the superclass of the current object and can be used to call superclass methods.
39. **switch** - Starts a switch statement to select one of many code blocks to execute.
40. **synchronized** - Used to ensure that a method or block of code is accessed by only one thread at a time.
41. **this** - Refers to the current object.
42. **throw** - Used to explicitly throw an exception.
43. **throws** - Indicates the exceptions that a method might throw.
44. **transient** - Prevents serialization of a variable.
45. **try** - Starts a block of code that will be tested for exceptions.
46. **void** - Specifies that a method does not return a value.
47. **volatile** - Indicates that a variable’s value will be modified by different threads.
48. **while** - Starts a while loop.
49. **record** - Used to define a record class in newer versions of Java (introduced in Java 14 as a preview feature).
50. **sealed** - Used to restrict which classes can subclass a class or implement an interface (introduced in Java 15).
51. **non-sealed** - Allows a class to be subclassed despite being in a sealed hierarchy (introduced in Java 17).
52. **permit** - Used in conjunction with sealed (introduced in Java 17).
53. **instanceof** - Tests if an object is an instance of a specific class or implements a specific interface.
54. **yield** - Used in switch expressions to return a value (introduced in Java 12).
55. **switch** - Used in switch expressions and statements (enhanced with new features in Java 12 and Java 14).

**Chapter 2**

**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

**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.

**Declaring a Class**

Every Java program consists of at least one class that you (the programmer) define. The class keyword introduces a class declaration and is immediately followed by the class name. Keywords in Java are always spelled with all lowercase letters. Every class we define begins with the public keyword. For now, we simply require public. You’ll learn more about public and non-public classes. Filename for a public Class must be placed in a file that has a filename of the form ClassName.java.

**Class Names and Identifiers**

By convention, class names begin with a capital letter and capitalize the first letter of each word. A class name is an identifier—a series of characters consisting of letters, digits, underscores (\_) and dollar signs ($) that does not begin with a digit and does not contain spaces. Some valid identifiers are

Welcome1,

$value,

\_value,

m\_inputField1 and

button7.

The name 7button is not a valid identifier because it begins with a digit, and the name input field is not a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not a class name. Java is case sensitive—uppercase and lowercase letters are distinct—so value and Value are different (but both valid) identifiers.

Class Body

A left brace {, begins the body of every class declaration. A corresponding right brace }, must end each class declaration.

**Method Declaration**

* **public static void main(String[] args)** is the method declaration that serves as the entry point of any Java application.
* **public**: This access modifier means the method can be accessed from outside the class. It's a convention for the main method so the Java Virtual Machine (JVM) can execute it.
* **static**: The keyword indicates that the method belongs to the class, rather than to instances of the class. This allows the JVM to call the main method without creating an object of the class.
* **void**: This indicates that the main method does not return any value. In other words, it doesn't give any output back to the system.
* **main**: This is the name of the method. The JVM specifically looks for a method named main to begin execution.
* **String[] args**: This is an array of String objects, which can hold command-line arguments passed to the program when it is run. These arguments are optional, and you can use them to provide input data when executing the program from the command line. (We'll discuss this in more detail in Chapter 6, as you mentioned.)

**Method Body**

* The **left curly brace {** marks the beginning of the method body, where the actual code to be executed resides.
* In this block, you'll define the steps or instructions the program will execute when the main method is called. In most simple Java programs, the body may contain just a few statements, such as printing output to the console.

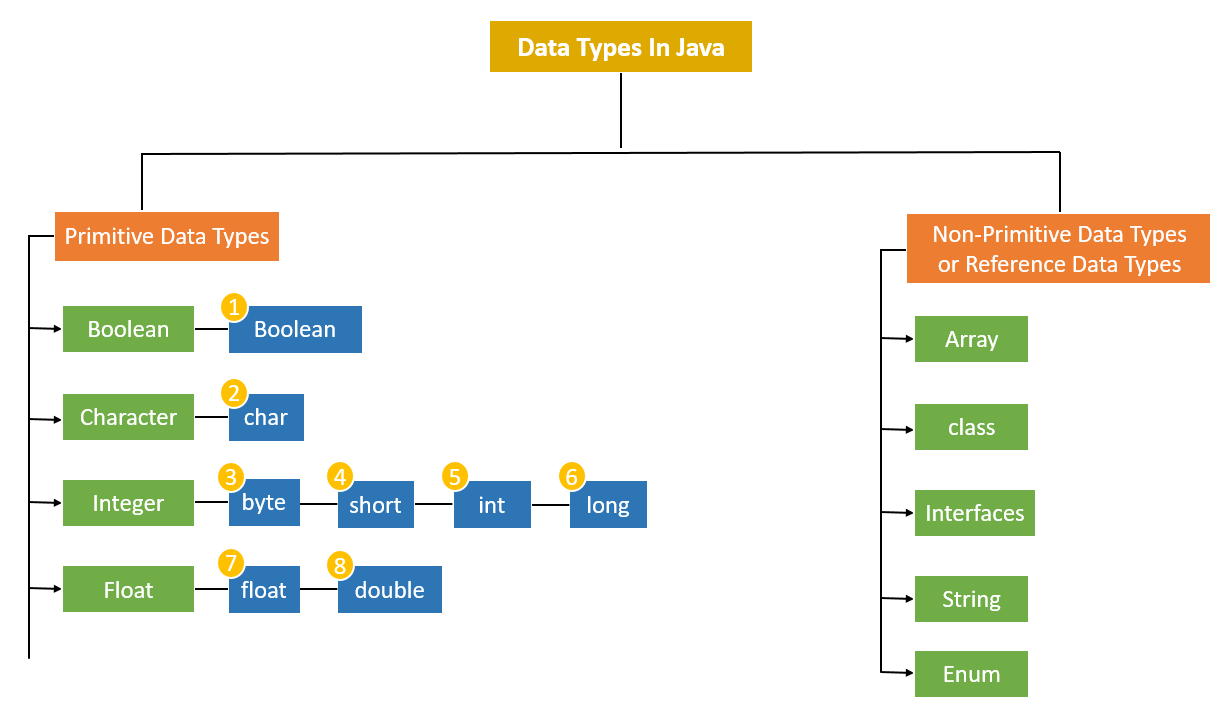
**Data types**

**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



**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. **short**: Represents a 16-bit signed integer.
   * Range: -32,768 to 32,767
   * Default value: 0
   * Example: short myShort = 500;
4. **int**: Represents a 32-bit signed integer.
   * Range: -2^31 to 2^31 - 1
   * Default value: 0
   * Example: int myInt = 100000;
5. **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.
* **Syntax Requirement**: It's a syntactic requirement in Java to use L (or l) when specifying long literals. If you omit the L, Java assumes it's an int, and you may encounter a compilation error if the number is too large for an int.
  + Example: long myLong = 100000L;

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.
* **Syntax Requirement**: It's a syntactic requirement in Java to use f (or F) when specifying float literals. If you omit the f, Java assumes it's a double, and you may encounter a compilation error if the number is not within the range or precision of float.
  + Example: float myFloat = 3.14f;

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;
2. **char**: Represents a single 16-bit Unicode character.
   * Default value: '\u0000' (null character)
   * Example: char myChar = 'A';

**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);

**Day 3**

**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**

**Input**

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(0) 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.
3. **nextBigInteger():** Reads the next token of input as a BigInteger. This method requires a String argument to specify the radix (e.g., "10" for decimal).
4. **nextBigDecimal():** Reads the next token of input as a BigDecimal.

**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**

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

**1. Arithmetic Operators**

Used for basic mathematical operations.

* **Addition (+)**: Adds two values.
  + **Example**: int sum = 5 + 3; // sum is 8
* **Subtraction (-)**: Subtracts the second value from the first.
  + **Example**: int difference = 10 - 4; // difference is 6
* **Multiplication (\*)**: Multiplies two values.
  + **Example**: int product = 7 \* 2; // product is 14
* **Division (/)**: Divides the first value by the second.
  + **Example**: int quotient = 8 / 2; // quotient is 4
* **Modulus (%)**: Returns the remainder of the division.
  + **Example**: int remainder = 9 % 4; // remainder is 1

**2. Comparison Operators**

Used to compare two values.

* **Equal to (==)**: Checks if two values are equal.
  + **Example**: boolean isEqual = (5 == 5); // isEqual is true
* **Not equal to (!=)**: Checks if two values are not equal.
  + **Example**: boolean isNotEqual = (5 != 3); // isNotEqual is true
* **Greater than (>)**: Checks if the first value is greater than the second.
  + **Example**: boolean isGreater = (7 > 3); // isGreater is true
* **Less than (<)**: Checks if the first value is less than the second.
  + **Example**: boolean isLess = (5 < 8); // isLess is true
* **Greater than or equal to (>=)**: Checks if the first value is greater than or equal to the second.
  + **Example**: boolean isGreaterOrEqual = (5 >= 5); // isGreaterOrEqual is true
* **Less than or equal to (<=)**: Checks if the first value is less than or equal to the second.
  + **Example**: boolean isLessOrEqual = (4 <= 7); // isLessOrEqual is true

**3. Logical Operators**

Used to perform logical operations.

* **Logical AND (&&)**: Returns true if both conditions are true.
  + **Example**: boolean result = (5 > 3) && (8 > 6); // result is true
* **Logical OR (||)**: Returns true if at least one condition is true.
  + **Example**: boolean result = (5 > 3) || (2 > 6); // result is true
* **Logical NOT (!)**: Reverses the logical state.
  + **Example**: boolean result = !(5 > 3); // result is false

**4. Assignment Operators**

Used to assign values to variables.

* **Simple Assignment (=)**: Assigns a value to a variable.
  + **Example**: int x = 10;
* **Add and Assign (+=)**: Adds a value to the variable and assigns the result.
  + **Example**: x += 5; // equivalent to x = x + 5
* **Subtract and Assign (-=)**: Subtracts a value from the variable and assigns the result.
  + **Example**: x -= 3; // equivalent to x = x - 3
* **Multiply and Assign (\*=)**: Multiplies the variable by a value and assigns the result.
  + **Example**: x \*= 4; // equivalent to x = x \* 4
* **Divide and Assign (/=)**: Divides the variable by a value and assigns the result.
  + **Example**: x /= 2; // equivalent to x = x / 2
* **Modulus and Assign (%=)**: Applies modulus operation and assigns the result.
  + **Example**: x %= 3; // equivalent to x = x % 3

**5. Bitwise Operators**

Operate on the binary representation of numbers.

* **Bitwise AND (&)**: Performs a bitwise AND.
  + **Example**: int result = 5 & 3; // result is 1
* **Bitwise OR (|)**: Performs a bitwise OR.
  + **Example**: int result = 5 | 3; // result is 7
* **Bitwise XOR (^)**: Performs a bitwise XOR.
  + **Example**: int result = 5 ^ 3; // result is 6
* **Bitwise NOT (~)**: Inverts all bits.
  + **Example**: int result = ~5; // result is -6
* **Left Shift (<<)**: Shifts bits to the left.
  + **Example**: int result = 5 << 1; // result is 10
* **Right Shift (>>)**: Shifts bits to the right.
  + **Example**: int result = 5 >> 1; // result is 2
* **Unsigned Right Shift (>>>)**: Shifts bits to the right, filling with zeros.
  + **Example**: int result = -5 >>> 1; // result is 2147483642

**6. Unary Operators**

Operate on a single operand.

* **Unary Plus (+)**: Indicates a positive value (usually optional).
  + **Example**: int x = +5; // x is 5
* **Unary Minus (-)**: Negates the value.
  + **Example**: int x = -5; // x is -5
* **Increment (++)**: Increases a value by 1.
  + **Example**: int x = 5; x++; // x is 6
* **Decrement (--)**: Decreases a value by 1.
  + **Example**: int x = 5; x--; // x is 4

**7. Conditional (Ternary) Operator**

A shorthand for an if-else statement.

* **Conditional (? :)**: Returns one of two values based on a condition.
  + **Example**: int max = (a > b) ? a : b; // max is a if a > b, otherwise max is b

**Operator precedence**

Operator precedence determines the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated before those with lower precedence. Understanding operator precedence is crucial for writing correct and predictable code.

Summary of Java operator precedence, from highest to lowest:

1. **Postfix**:
   * expr++ (post-increment)
   * expr-- (post-decrement)
2. **Unary**:
   * + (unary plus)
   * - (unary minus)
   * ++ (pre-increment)
   * -- (pre-decrement)
   * ! (logical NOT)
   * ~ (bitwise NOT)
3. **Multiplicative**:
   * \* (multiplication)
   * / (division)
   * % (modulus)
4. **Additive**:
   * + (addition)
   * - (subtraction)
5. **Shift**:
   * << (left shift)
   * >> (right shift)
   * >>> (unsigned right shift)
6. **Relational**:
   * < (less than)
   * <= (less than or equal to)
   * > (greater than)
   * >= (greater than or equal to)
   * instanceof (type comparison)
7. **Equality**:
   * == (equal to)
   * != (not equal to)
8. **Bitwise AND**:
   * & (bitwise AND)
9. **Bitwise XOR**:
   * ^ (bitwise XOR)
10. **Bitwise OR**:
    * | (bitwise OR)
11. **Logical AND**:
    * && (logical AND)
12. **Logical OR**:
    * || (logical OR)
13. **Ternary**:
    * ? : (ternary conditional)
14. **Assignment**:
    * = (assignment)
    * += (addition assignment)
    * -= (subtraction assignment)
    * \*= (multiplication assignment)
    * /= (division assignment)
    * %= (modulus assignment)
    * <<= (left shift assignment)
    * >>= (right shift assignment)
    * >>>= (unsigned right shift assignment)
    * &= (bitwise AND assignment)
    * ^= (bitwise XOR assignment)
    * |= (bitwise OR assignment)
15. **Comma**:
    * , (comma operator)

**Chapter 3**

**Day 4**

**Control structures**

A program is a list of instructions. **Control structures are programming blocks that can change the path we take through those instructions.**

**There are three kinds of control structures:**

* **Decision making or Conditional Branches:** They are used for choosing between two or more paths. There are three types in Java:
* Decision Making statements
  + if statements
  + switch statement

### 1. Conditional Statements

**a. if Statement:** Used to execute a block of code if a specified condition is true.

if (condition) {

// code to be executed if the condition is true

}

**b. if-else Statement:** Provides an alternative block of code that is executed if the condition is false.

if (condition) {

// code to be executed if the condition is true

} else {

// code to be executed if the condition is false

}

**c. else if Statement:** Allows for multiple conditions to be checked in sequence.

if (condition1) {

// code to be executed if condition1 is true

} else if (condition2) {

// code to be executed if condition2 is true

} else {

// code to be executed if none of the conditions are true

}

**d. switch Statement:** Used to execute one block of code among many based on the value of an expression.

switch (expression) {

case value1:

// code to be executed if expression equals value1

break;

case value2:

// code to be executed if expression equals value2

break;

// more cases

default:

// code to be executed if no cases match

}

* **Loops statement**: They are used to iterate through multiple values/objects and repeatedly run specific code blocks.
* Loop statements
  + do while loop
  + while loop
  + for loop
  + for-each loop

### Looping Statements

**a. for Loop:** Used for iterating a block of code a specified number of times.

for (initialization; condition; update) {

// code to be executed repeatedly

}

**b. while Loop:** Executes a block of code as long as a specified condition is true.

while (condition) {

// code to be executed repeatedly

}

**c. do-while Loop:** Similar to the while loop, but guarantees that the block of code will be executed at least once.

do {

// code to be executed repeatedly

} while (condition);

* **Jump Statements or Branching Statements:** They are used to alter the flow of control in loops. There are two types in Java:
* Jump statements
  + break statement
  + continue statement

### Jump Statements

**a. break:** 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

}

**c. return:** Exits from the current method and optionally returns a value.

public int sum(int a, int b) {

return a + b; // exit the method and return the result

}

**Week 2**

**Day 1**

• Sentinel controlled repetition

• Nested control statements

• Compound assignment operators

• Increment / decrement operators

Chapter 4

Day 2

• For repetition statement

• Do while repetition statement

• Switch selection statement

Day 3

• Switch selection statement cont’d

• Break and continue statements

• Logical operators

Day 4

Catch up gap

Week 3

Chapter 5

Day 1

• Declaring and calling methods

Day 2

• Static methods (class Math)

• Method call stack

• Argument promotion

• Scope of declarations

Day 3

• Method overloading

• Random number generation

• Introduction to enums\*

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 creating 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

**Class Arrays**

In Java, Class Arrays helps you avoid reinventing the wheel by providing static methods for common array manipulations. the Arrays class is a utility class provided in the java.util package. It contains various static methods that are used to manipulate arrays, such as sorting, searching, and converting arrays to strings. 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]]"

**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**

The Java API provides several predefined data structures, called collections, used to store groups of related objects in memory. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This reduces application-development time. You’ve used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements. The collection class **ArrayList<T>** (package java.util) provides a convenient solution to this problem—it can dynamically change its size to accommodate more elements.

The **T** (by convention) is a placeholder—when declaring a new ArrayList, replace it with the type of elements that you want the ArrayList to hold.

For example, ArrayList <String> list = new ArrayList<>();

declares list as an ArrayList collection that can store only Strings.

Classes with this kind of placeholder that can be used with any type are called generic classes. Only nonprimitive types can be used to declare variables and create objects of generic classes. However, Java provides a mechanism—known as boxing—that allows primitive values to be wrapped as objects for use with generic classes. So, for example,

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

declares integers as an ArrayList that can store only Integers. When you place an int value into an ArrayList, the int value is boxed (wrapped) as an Integer object, and when you get an Integer object from an ArrayList, then assign the object to an int variable, the int value inside the object is unboxed (unwrapped).

**Method and their Description**

ArrayList is a part of the java.util package and provides a resizable array implementation of the List interface. It allows for dynamic resizing, which means that you don’t need to specify the size of the list in advance. 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.
   * removeAll(Collection<?> c): Removes all of the elements in the list that are also contained in the specified collection.
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.

**Chapter 7**

**Day 3**

Object-Oriented Programming (OOP) in Java is a programming paradigm 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.

OOP in Java is built on four fundamental principles:

1. **Encapsulation**:
2. **Inheritance**:
3. **Polymorphism**:
4. **Abstraction**:

These principles work together to create a structured and modular approach to programming, making it easier to manage and scale complex applications. Java’s strong support for OOP helps developers create reusable and maintainable code.

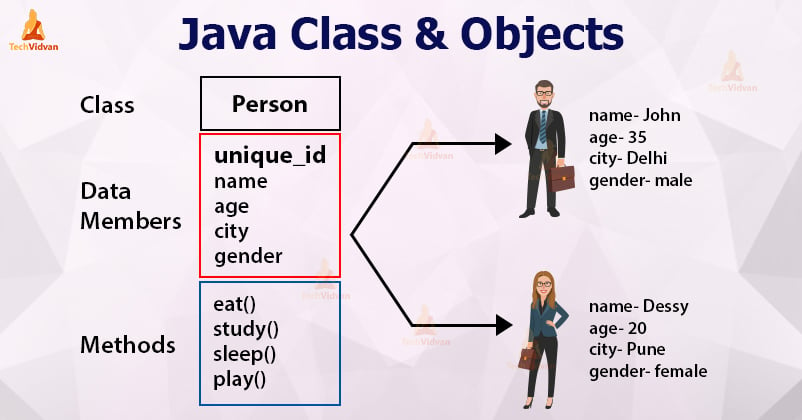
**Classes and objects**

**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().



**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 owner name.

**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.

**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
2. One-to-many
3. Many-to-one
4. Many-to-many.
5. For example, a person can have only one passport. That is a “**one-to-one**” relationship.
6. If we talk about the association between a Bank and Employee, a bank can have many employees, so it is a “**one-to-many**” relationship.
7. Similarly, every city exists in exactly one state, but a state can have many cities, which is a “**many-to-one**” relationship.
8. Lastly, if we talk about the association between a teacher and a student, multiple students can be associated with a single teacher and a single student can also be associated with multiple teachers but both can be created or deleted independently. This is a “**many-to-many**” relationship.

**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**

1. **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.
2. **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.
3. **Implementation**: Aggregation is implemented using a reference to another object. The child object can be shared among multiple parents.
4. **Example**: Consider a Library and a Book. A library has books, but a book can exist without a library and be part of different libraries.

### 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.

1. **Author and Books**:

An **Author** writes **Books**. An author can write multiple books, and books are not dependent on any specific author for their existence.

1. **Car and Wheels**:

A **Car** uses **Wheels**. While a car needs wheels to function, wheels themselves can be swapped or used on different cars.

1. **Event and Attendees**:

An **Event** has **Attendees**. People can attend multiple events and exist independently of any single event.

1. **Project and Team Members**:

A **Project** involves **Team Members**. Team members can be part of multiple projects over time.

1. **Restaurant and Menu Items**:

A **Restaurant** offers **Menu Items**. Menu items can be changed or moved between different restaurants.

**Composition**

1. **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.
2. **Lifetime**: In composition, if the parent object is destroyed, the child object is also destroyed. The child object cannot exist without the parent.
3. **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.
4. **Example**: Consider a House and Room. A house has rooms, and rooms cannot exist without a house.

### Examples of Composition

1. **House and Rooms:**

A House is made up of Rooms. Rooms cannot exist separately from a house; if the house is demolished, the rooms are gone too.

1. **Human and Heart:**

A Human has a Heart. The heart is an integral part of a human and cannot exist separately in this context.

1. **Computer and Motherboard:**

A Computer contains a CPU (Central Processing Unit). The CPU is essential to the computer's operation and is typically not used independently.

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. **Body and Cells:**

A Body is made up of Cells. Cells cannot exist outside the body in a functional way as part of an organism.

1. **Computer and Hard Drive:**

A Computer includes a Hard Drive. The hard drive is a crucial part of the computer’s storage system.

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.

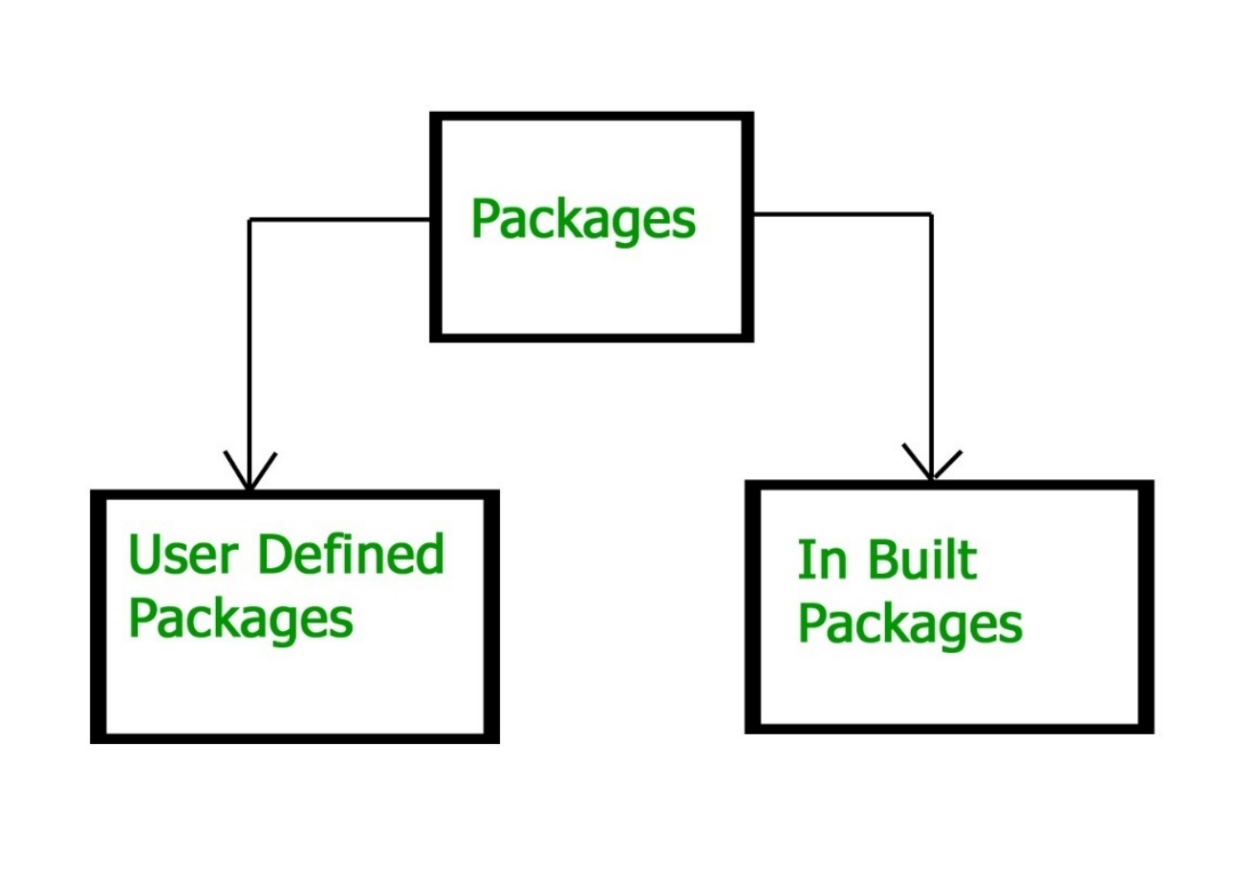
**Package**

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 in Java 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.**

Why use inheritance in java

* For Method Overriding
* For Code Reusability.

**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**

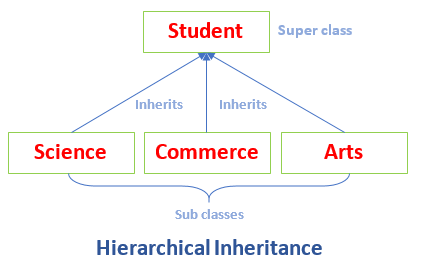
### 1. Single Inheritance

### 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



using the employee payroll example:

1. **Superclass: CommissionEmployee**
   * Inherits from Object.
   * Contains private variables: first name, last name, social security number, commission rate, and gross sales amount.
   * This class models employees who earn a percentage of their sales.
2. **Subclass: BasePlusCommissionEmployee (Initial Version)**
   * Also inherits from Object.
   * Declares its own private variables: first name, last name, social security number, commission rate, gross sales amount, and base salary.
   * Writing this class requires duplicating a lot of code from CommissionEmployee.
3. **Subclass: BasePlusCommissionEmployee (Inheriting from CommissionEmployee)**
   * Now, this subclass extends CommissionEmployee.
   * This design means it inherits the properties and behaviors of CommissionEmployee.
   * Attempting to access private variables from CommissionEmployee results in compilation errors because private members can’t be accessed by subclasses.
4. **Using Protected Variables**
   * If the variables in CommissionEmployee are declared as protected, BasePlusCommissionEmployee can directly access them.
   * This makes the subclass easier to implement since it can reuse the superclass’s properties without extra code.
5. **Encapsulation and Public Methods**
   * Finally, if CommissionEmployee keeps its variables private, BasePlusCommissionEmployee can still manipulate them through public methods provided by CommissionEmployee.
   * This approach maintains good software engineering practices by protecting the data while still allowing the subclass to function correctly.

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.

Polymorphism in object-oriented programming allows for writing generalized code that can handle different object types uniformly.

In the example of simulating animal movements, we have a superclass Animal with subclasses Fish, Frog, and Bird, each implementing a move method. An array of Animal references can contain different animal types, and when the move method is called on each object, each subclass responds according to its specific movement logic—fish swim, frogs jump, and birds fly. This demonstrates how the same method call can produce different behaviors based on the object's type, highlighting the core concept of polymorphism.

**Implementing for Extensibility**

Polymorphism enables systems to be easily extensible, allowing new classes to be added with minimal modifications to existing code. When a new class, such as Tortoise, is created as a subclass of Animal, it can seamlessly integrate into the program. The only required changes involve defining the Tortoise class and updating the simulation to instantiate a Tortoise object. The general parts of the program that handle movement for all Animal objects remain unchanged, demonstrating how polymorphism simplifies the extension of functionality without disrupting existing code.

**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 behaviour 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**

In Java, an **interface** 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.

**Case Study: Payroll System Using Polymorphism (Done by Faculty)**

**Overview of the Payroll System**

Case study of the Payroll System using polymorphism, focusing on the employee hierarchy and how it functions within the application.

The company needs to manage payroll for four different types of employees, each with unique payment structures. By using polymorphism and an abstract class, the system can handle these variations efficiently.

**Employee Types**

1. **Salaried Employee**:
   * **Payment Structure**: Fixed weekly salary, regardless of hours worked.
   * **Example**: A manager or full-time employee who earns a set amount each week.
2. **Hourly Employee**:
   * **Payment Structure**: Paid per hour worked, with overtime pay (1.5 times the hourly rate) for hours over 40.
   * **Example**: A technician or part-time worker.
3. **Commission Employee**:
   * **Payment Structure**: Paid a percentage based on sales made, without any base salary.
   * **Example**: A salesperson who earns commissions on sales.
4. **Base Plus Commission Employee**:
   * **Payment Structure**: Receives a base salary plus a commission on sales.
   * **Example**: A salesperson with a fixed salary plus additional earnings from sales.

**Abstract Class: Employee**

* **Purpose**: To define a general contract for all employee types. It provides a common interface and shared functionality.
* **Abstract Method**:
  + The class will include an abstract method, like earnings(), which must be implemented by all subclasses. This ensures that each type of employee has a way to calculate their pay based on their unique structure.

**Employee (Abstract Class)**

* **Attributes**:
  + private String firstName
  + private String lastName
  + private String socialSecurityNumber
* **Methods**:
  + abstract double earnings()
  + public String toString()
  + getter methods for the attributes.

**Classes Extending Employee**

1. **SalariedEmployee**:
   * Implements earnings() to return the fixed salary.
   * toString(): Returns a string with employee details
2. **HourlyEmployee**:
   * Implements earnings() to calculate the pay based on hours worked, factoring in overtime.
   * toString(): Returns a string with employee details.
3. **CommissionEmployee**:
   * Implements earnings() to compute pay based on sales figures and commission rate.
   * toString(): Returns a string with employee details.
4. **BasePlusCommissionEmployee**:
   * Inherits from CommissionEmployee and overrides calculatePay() to add a base salary plus commission. Also includes functionality to apply a salary increase (e.g., adding 10% to the base salary).
   * toString(): Returns a string with employee details.

### Employee Types Summary

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee Type** | **Attributes** | **Earnings Calculation** | **toString Output** |
| **Employee** | firstName | Abstract | firstName lastName social security number: SSN |
|  | lastName |  |  |
|  | social security number: SSN |  |  |
| **Salaried Employee** | - firstName: String | weeklySalary | salaried employee: firstName lastName social security number: SSN  weekly salary: weeklySalary |
|  | - lastName: String |  |  |
|  | - socialSecurityNumber: String |  |  |
|  | - weeklySalary: double |  |  |
| **Hourly Employee** | - firstName: String | if (hours <= 40) wage \* hours else if (hours > 40) { 40 \* wage + ( hours - 40 ) \* wage \* 1.5 } | " hourly employee: firstName lastName social security number: SSN  hourly wage: wage; hours worked: hours” |
|  | - lastName: String |  |  |
|  | - socialSecurityNumber: String |  |  |
|  | - wage: double |  |  |
|  | - hours: double |  |  |
| **Commission Employee** | - firstName: String | commissionRate \* grossSales | " commission employee: firstName lastName social security number: SSN gross sales: grossSales  commission rate: commissionRate” |
|  | - lastName: String |  |  |
|  | - socialSecurityNumber: String |  |  |
|  | - grossSales: double |  |  |
|  | - commissionRate: double |  |  |
| **Base Plus Commission Employee** | - firstName: String | (commissionRate \* grossSales) + baseSalary | " base salaried commission employee: firstName lastName  social security number: SSN  gross sales: grossSales;  commission rate: commissionRate;  base salary: baseSalary" |
|  | - lastName: String |  |  |
|  | - socialSecurityNumber: String |  |  |
|  | - grossSales: double |  |  |
|  | - commissionRate: double |  |  |
|  | - baseSalary: double |  |  |

**Polymorphism in Action**

* **Polymorphic Payroll Calculation**:
  + The payroll application can maintain a list of Employee objects, regardless of their specific types. When it iterates through this list and calls calculatePay(), the appropriate method for each specific employee type is invoked.
  + This allows the application to handle various employee types seamlessly without needing to know the specifics of each type at compile time.

**Benefits of Using Polymorphism**

* **Extensibility**: If a new employee type is needed in the future (e.g., a contractor), it can easily be added by creating a new subclass that implements the calculatePay() method.
* **Maintainability**: Changes to the pay calculation logic in one employee type won’t affect others, as each class manages its own logic.
* **Code Reusability**: Common functionality can be centralized in the Employee class, reducing code duplication across subclasses.

**Summary**

In summary, this payroll system case study effectively demonstrates how polymorphism and abstract classes can be used to handle a diverse range of employee payment structures. By leveraging these OOP principles, the system remains flexible, maintainable, and easy to extend, accommodating changes in business requirements or new employee types with minimal effort.

**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?**

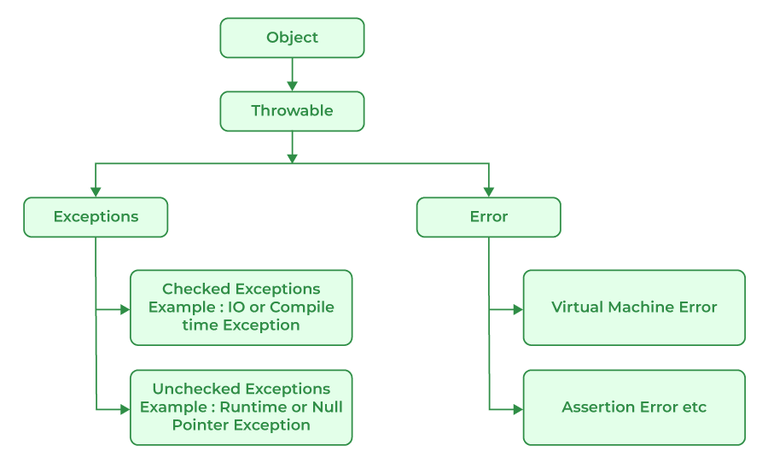
Errors represent serious problems that are typically outside the control of the application or the programmer. They indicate conditions that the application usually cannot recover from. and we should not try to handle errors

### 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**:

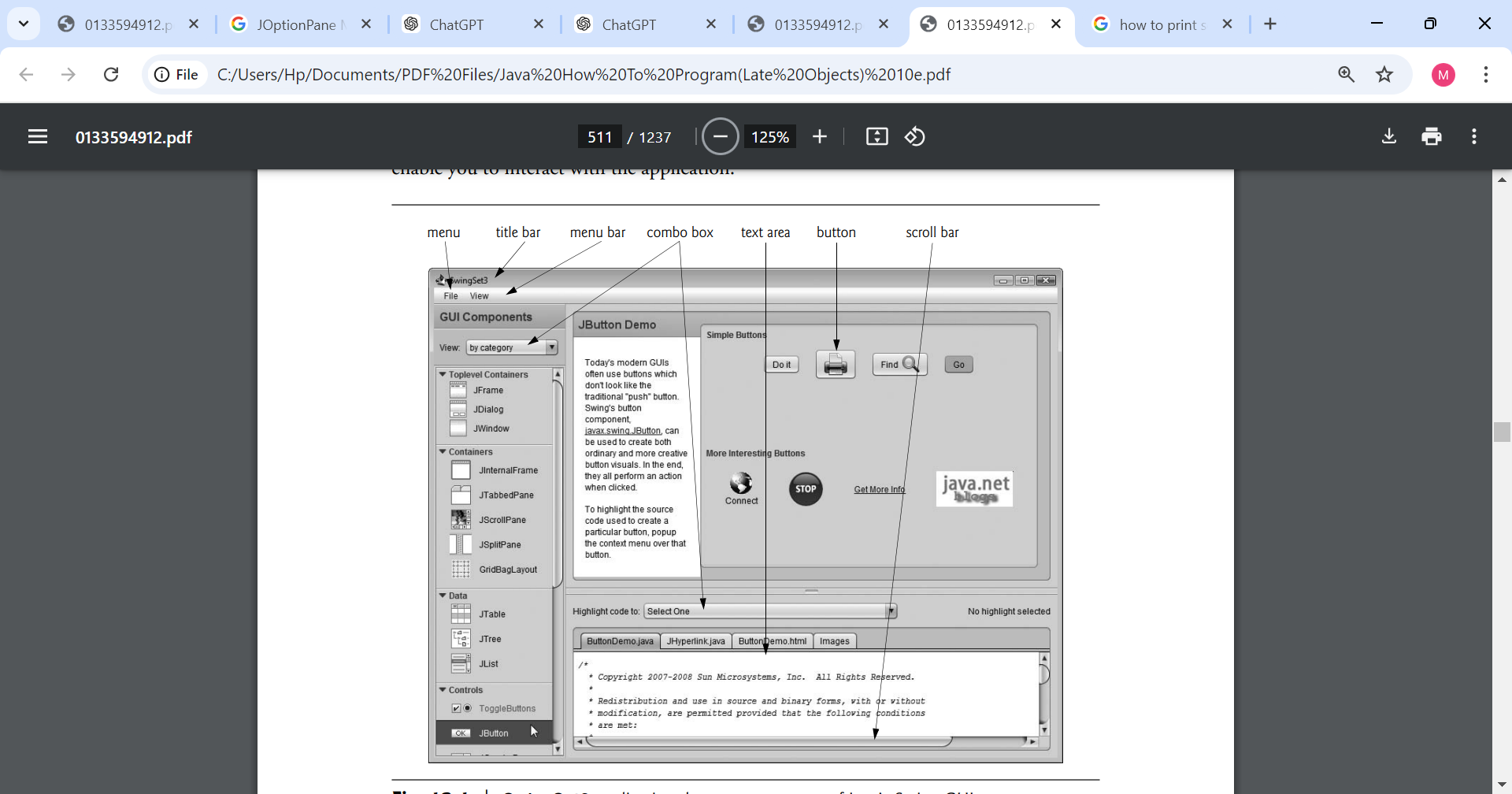
A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. These are the building blocks of a GUI. Common examples include buttons, text fields, labels, and sliders. Each component allows users to perform specific actions, such as entering data or triggering events.

**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();

}

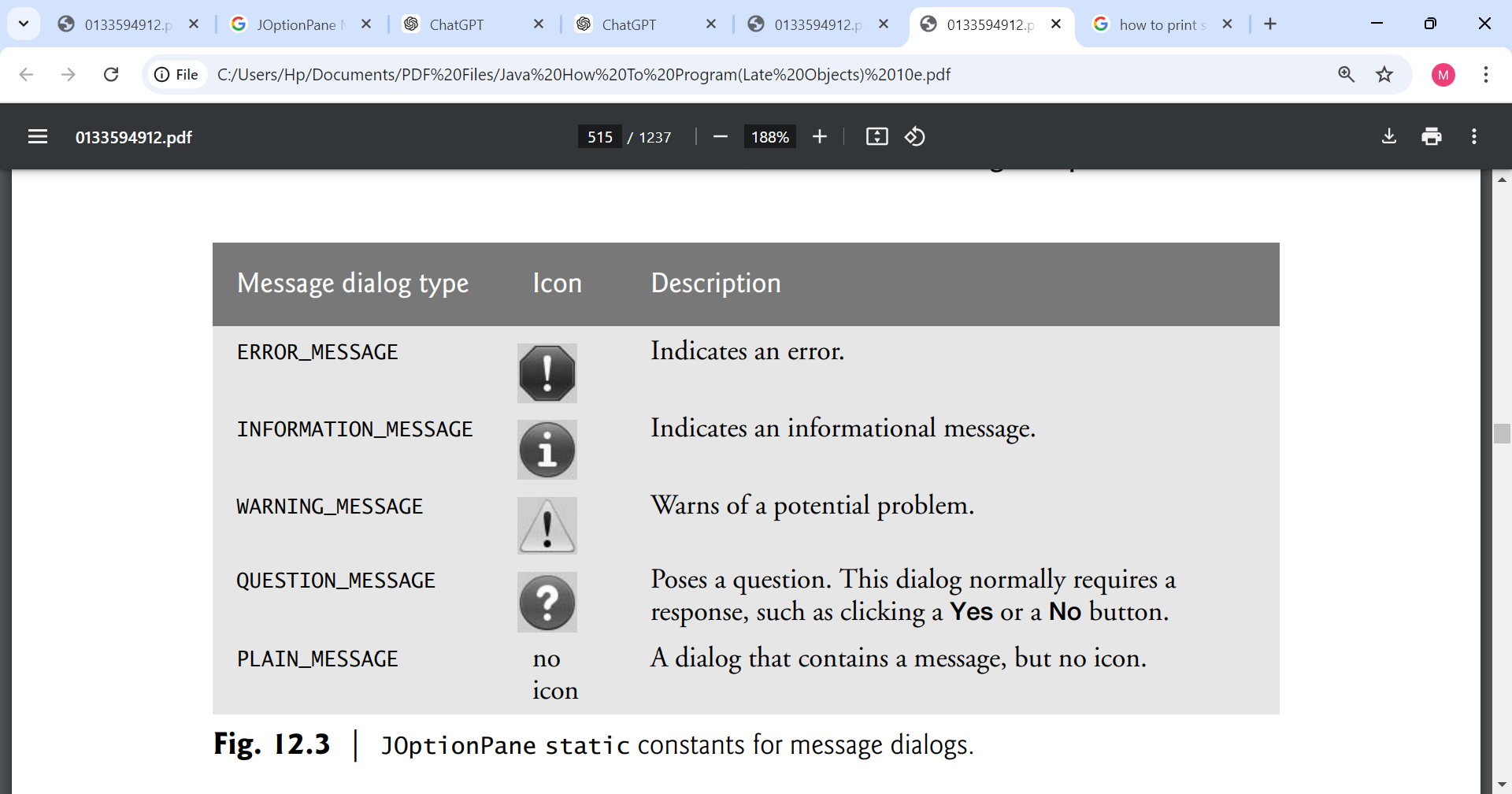
* + This code should be executed before creating any GUI components to ensure that the look-and-feel is applied.

### 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.**

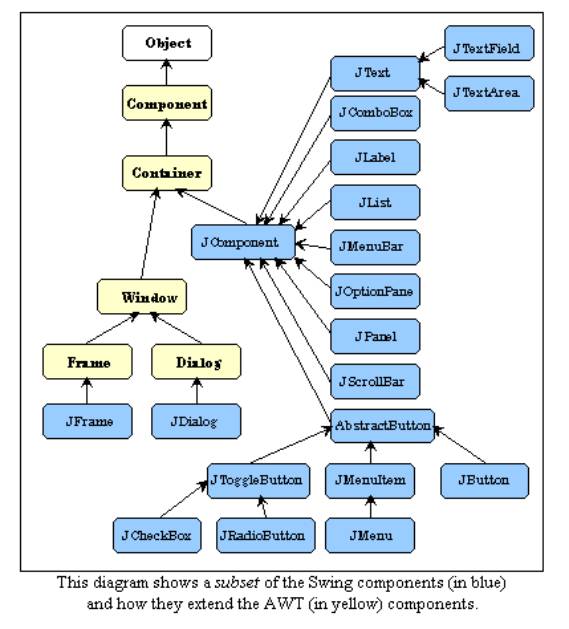
1. **Native Look and Feel**:
   * AWT components are heavyweight, meaning they rely on the native operating system’s GUI components. This makes them look and behave like standard components of the host OS. For instance, a button created with AWT on Windows will have the same appearance as other Windows 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**

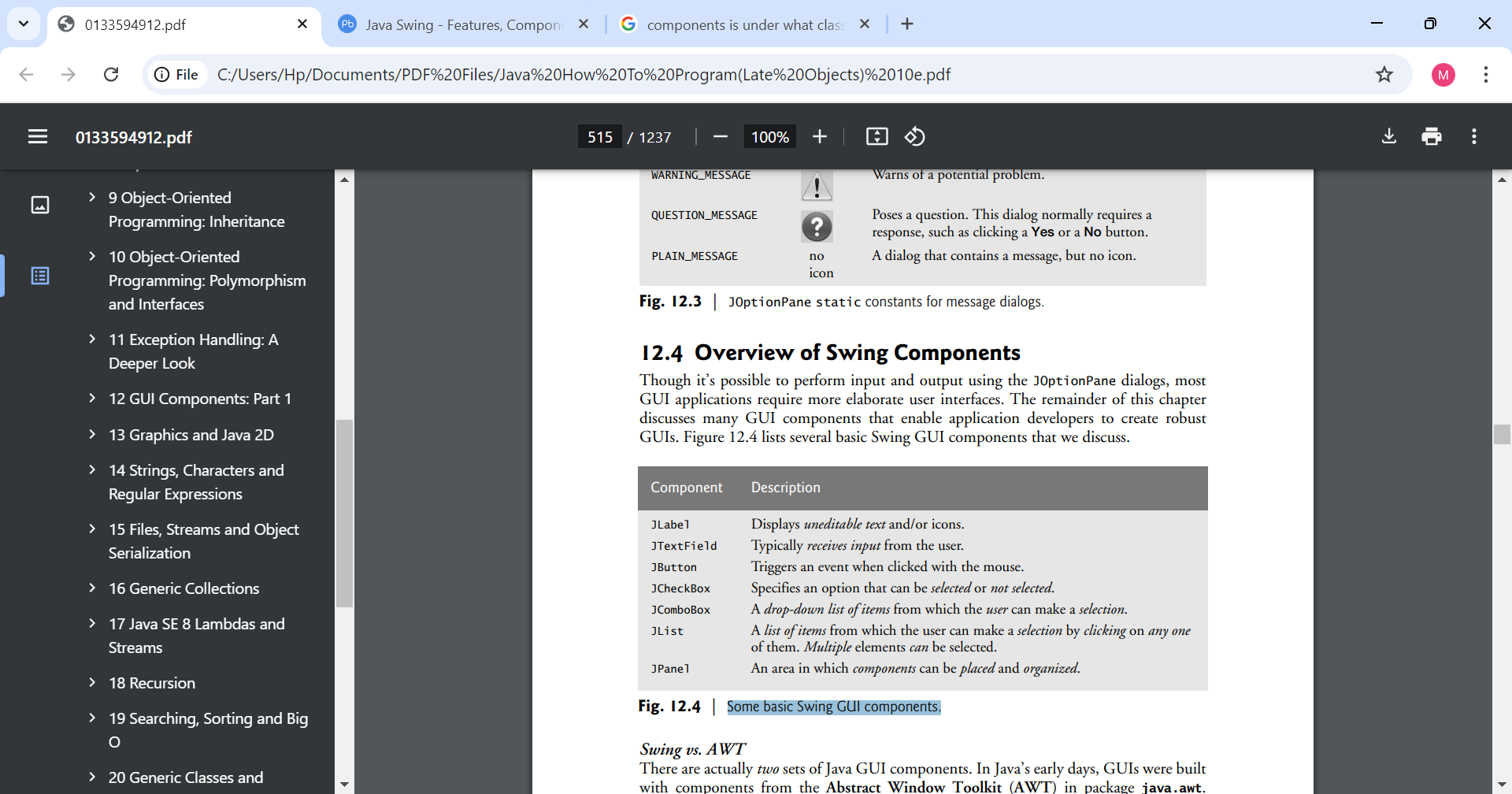
1. **Lightweight Components**:
   * Swing components are lightweight, meaning they are written entirely in Java and do not rely on native GUI components. This allows for a consistent look and feel across all platforms, making it easier to develop cross-platform applications.
2. **Customizable Look and 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**



**Example of using JFrame, JLabel, and ImageIcon classes**

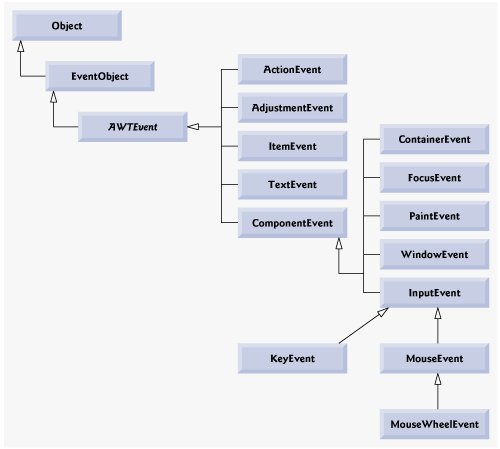
**What is an Event and Event Handler?**

* **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. 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 (like pressing the Enter key), the corresponding event handler executes to perform a task. The process of responding to events is known as event handling.

**Example of using JTextField, JPasswordField and Event Handling**

**Common GUI Event Types and 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**

**Strings, Characters 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.

**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"

**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.
2. **charAt(int index)**: Returns the character at the specified index.
3. **substring(int beginIndex)**: Returns a substring starting from the specified index.
4. **substring(int beginIndex, int endIndex)**: Returns a substring from beginIndex to endIndex - 1.
5. **indexOf(String str)**: Returns the index of the first occurrence of the specified substring.
6. **toLowerCase()**: Converts the string to lower case.
7. **toUpperCase()**: Converts the string to upper case.
8. **trim()**: Removes leading and trailing whitespace.
9. **replace(char oldChar, char newChar)**: Replaces all occurrences of a specified character with a new character.
10. **split(String regex)**: Splits the string into an array based on the specified

**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.

• Assignment: class Character, StringBuilder, StringBuffer

**Class Character**

* **Purpose**: The Character class is a wrapper for the primitive data type char. It provides methods to manipulate and convert characters.
* **Key Features**:
  + **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.
  + **Character Properties**: Can retrieve information about characters, such as their Unicode value and whether they are letters, digits, or whitespace.
* **Usage**: Often used when you need to perform character-level manipulations or checks.

**Class StringBuilder**

* **Purpose**: StringBuilder is used to create mutable (modifiable) strings. It allows you to build and modify strings efficiently.
* **Key Features**:
  + **Mutable**: Unlike String, which is immutable, StringBuilder can be changed without creating a new object.
  + **Performance**: More efficient than using String for concatenation in loops or when making many modifications, as it reduces memory overhead.
  + **Common Methods**:
    - append(String str): Adds a string to the end.
    - insert(int offset, String str): Inserts a string at a specified position.
    - delete(int start, int end): Removes a substring.
    - reverse(): Reverses the string.
    - toString(): Converts the StringBuilder object to a String.
* **Usage**: Ideal for scenarios where you need to manipulate strings frequently, such as constructing strings in loops.

**Class StringBuffer**

* **Purpose**: Similar to StringBuilder, StringBuffer is also used to create mutable strings but is synchronized, making it thread-safe.
* **Key Features**:
  + **Thread-Safe**: Methods are synchronized, which makes it suitable for use in multi-threaded environments.
  + **Mutable**: Like StringBuilder, it can be modified without creating new objects.
  + **Common Methods**: Shares many methods with StringBuilder, such as append(), insert(), delete(), and toString().
* **Usage**: Use StringBuffer when working in a multi-threaded context where multiple threads might modify the same string.

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

### append Method

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.

### Usage

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()**:

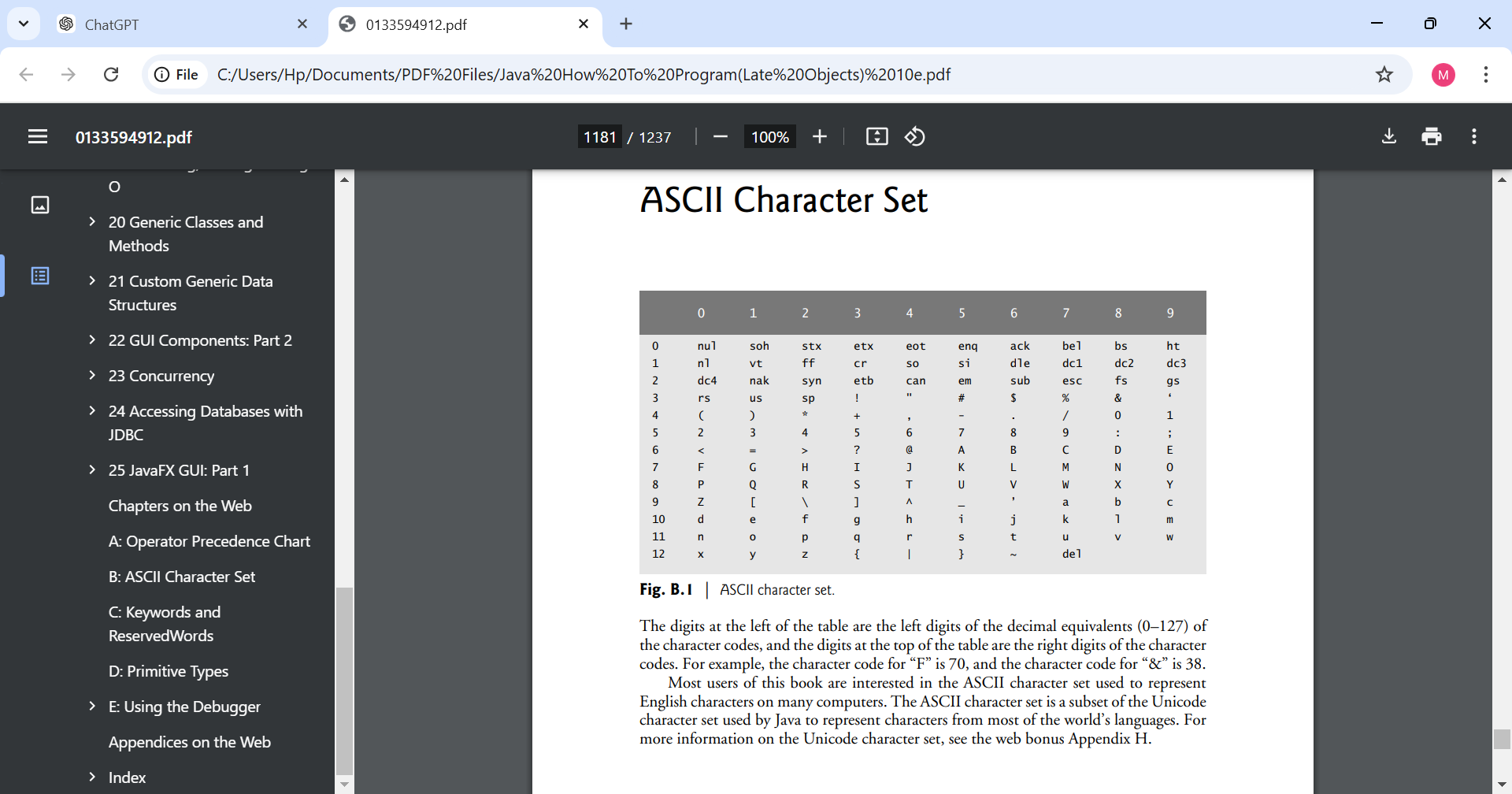
Compares two strings lexicographically. Use this when you need to determine the order of strings (i.e., sorting).

**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**

* **Definition**: Regex is a sequence of characters that forms a search pattern. It’s often used for string matching and manipulation.
* **Applications**: Regex can be useful tool in form validation, searching text, and data parsing.

**2. Basic Syntax**

* **Characters**: Start with basic characters like letters, digits, and punctuation.
* **Metacharacters**:
  + . (any character)
  + ^ (start of a line)
  + $ (end of a line)
  + \* (zero or more)
  + + (one or more)
  + ? (zero or one)
  + [] (character classes)
  + | (alternation)
  + () (Capturing group)

**3. Java 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..

More Matches expressions

* ^ Matches beginning of line.
* $ Matches end of line.
* . Matches any single character except newline. Using m option allows it to match newline as well.
* [...] Matches any single character in brackets.
* [^...] Matches any single character not in brackets.
* \A Beginning of entire string.
* \z End of entire string.
* \Z End of entire string except allowable final line terminator.
* re\* Matches 0 or more occurrences of preceding expression.
* re+ Matches 1 or more of the previous thing.
* re? Matches 0 or 1 occurrence of preceding expression.
* re{n} Matches exactly n number of occurrences of preceding expression.
* re{n,} Matches n or more occurrences of preceding expression.
* re{n,m} Matches at least n and at most m occurrences of preceding expression.
* a|b Matches either a or b.
* re Groups regular expressions and remembers matched text.
* ?
* Groups regular expressions without remembering matched text.
* ? > re Matches independent pattern without backtracking.
* \w Matches word characters.
* \W Matches nonword characters.
* \s Matches whitespace. Equivalent to [\t\n\r\f].
* \S Matches nonwhitespace.
* \d Matches digits. Equivalent to [0-9].
* \D Matches nondigits.
* \A Matches beginning of string.
* \Z Matches end of string. If a newline exists, it matches just before newline.
* \z Matches end of string.
* \G Matches point where last match finished.
* \n Back-reference to capture group number "n".
* \b Matches word boundaries when outside brackets. Matches backspace 0x08 when inside brackets.
* \B Matches nonword boundaries.
* \n, \t, etc. Matches newlines, carriage returns, tabs, etc.
* \Q Escape quote all characters up to \E.
* \E Ends quoting begun with \Q.

• Pattern and Matcher Day 2 N/A

• 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)

**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**

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.

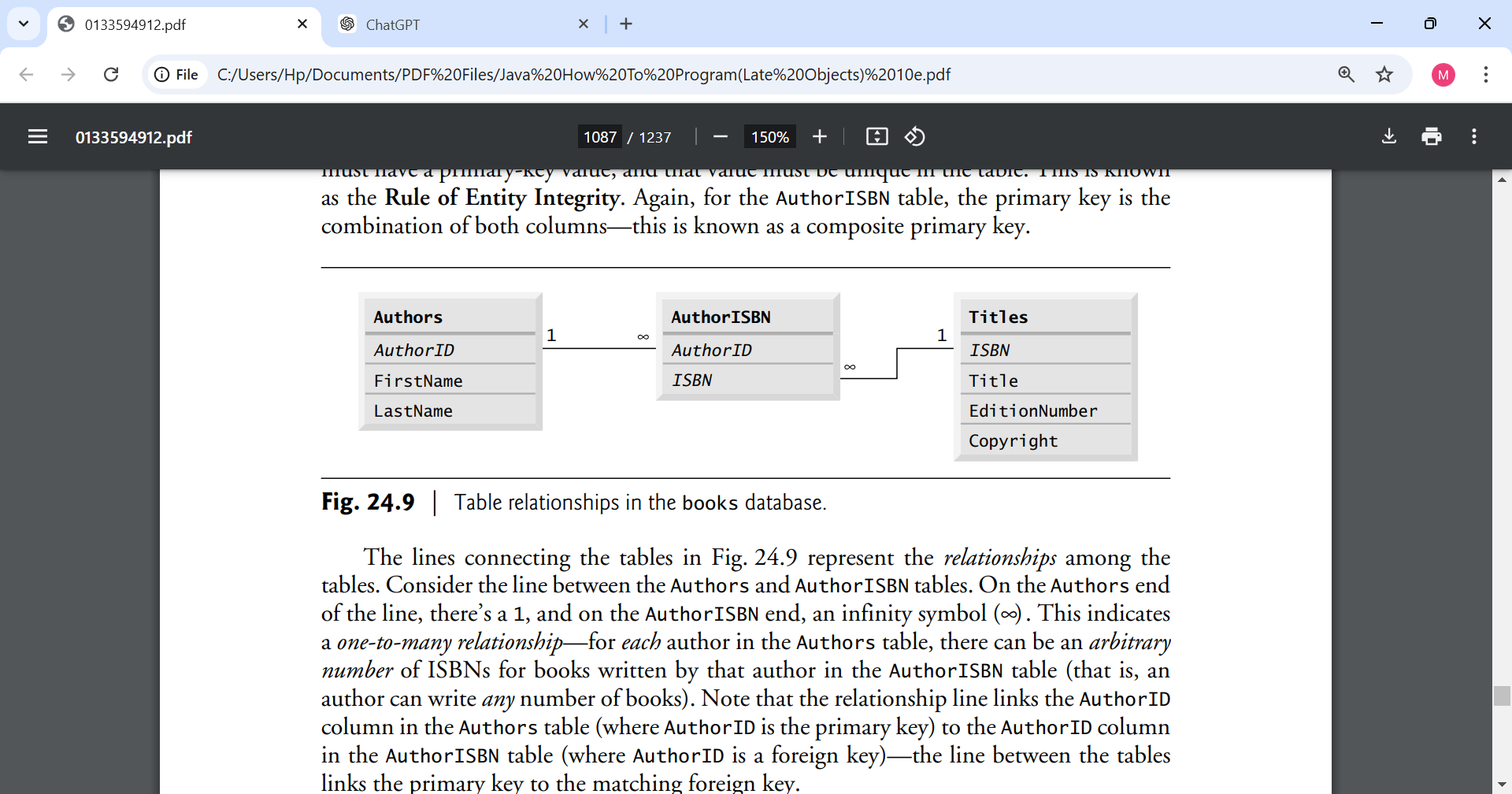
### Entity-Relationship (ER) Diagram

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