java1

☕ Java Notes

Made By : Kashif Sayyad

🟡 Module I — Introduction to Java & Basics

🔹 Object Oriented Programming Concepts

Object Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** rather than functions and logic. An object is a real-world entity that has **state** (attributes/properties) and **behavior** (methods/functions). OOP makes code more modular, reusable, and easier to maintain compared to procedural programming. The four core pillars of OOP are **Encapsulation** (bundling data and methods together and hiding internal details), **Inheritance** (a class acquiring properties and behaviors of another class), **Polymorphism** (one interface, many implementations — same method behaves differently based on context), and **Abstraction** (hiding complex implementation details and showing only essential features). OOP models real-world entities naturally — for example, a `Car` object has attributes like `color` and `speed`, and behaviors like `accelerate()` and `brake()`. Java is a purely object-oriented language where everything (except primitive types) is an object.

OOP Pillar	Description	Java Keyword
Encapsulation	Data hiding with getters/setters	`private`, `public`
Inheritance	Child class inherits parent	`extends`
Polymorphism	Same method, different behavior	`@Override`
Abstraction	Hide complexity, show essentials	`abstract`, `interface`

🔹 History of Java

Java was created by **James Gosling** and his team at **Sun Microsystems** in the early 1990s, initially as part of a project called **Green Project** aimed at developing software for consumer electronics. The language was originally called **Oak** (after an oak tree outside Gosling's office) but was later renamed **Java** (inspired by Java coffee). Java 1.0 was officially released to the public in **1995** with the revolutionary promise of **"Write Once, Run Anywhere (WORA)"** — meaning Java code compiled once could run on any platform with a Java Virtual Machine (JVM). Sun Microsystems was acquired by **Oracle Corporation in 2010**, and Oracle continues to develop and maintain Java today. Java quickly became one of the world's most popular programming languages, dominating enterprise application development, Android app development, and web backend systems. Major milestones include Java 5 (generics, autoboxing), Java 8 (lambda expressions, streams), and Java 17 (LTS version with modern features).

◆ Features of Java

Java's widespread adoption is largely due to its rich set of features that make it powerful, reliable, and platform-independent. **Platform Independent** — Java code is compiled into **bytecode** which runs on the JVM, making it OS-independent. **Object Oriented** — Everything in Java is an object, promoting modular and reusable code design. **Simple** — Java has a clean syntax inspired by C++ but removes complex features like pointers and multiple inheritance. **Secure** — Java runs inside the JVM sandbox, preventing direct access to memory and providing a secure execution environment. **Robust** — Strong type checking, exception handling, and garbage collection make Java programs reliable and less prone to crashes. **Multithreaded** — Java has built-in support for multithreading, allowing concurrent execution of multiple parts of a program. **High Performance** — Just-In-Time (JIT) compiler converts bytecode to native machine code at runtime for improved performance. **Distributed** — Java supports distributed computing through technologies like RMI and CORBA.

◆ Data Types in Java

Java is a **strongly typed** language, meaning every variable must be declared with a specific data type before use. Java has two categories of data types — **Primitive** and **Non-Primitive (Reference)** types. Primitive types are the most basic and are stored directly in memory — there are exactly 8 primitive types in Java. Non-primitive types include **String**, **Arrays**, **Classes**, and **Interfaces** — they store references (memory addresses) to objects rather than the actual values. The size and range of primitive types are fixed and platform-independent in Java, unlike in C/C++.

Data Type	Size	Default Value	Range
`byte`	1 byte	0	-128 to 127
`short`	2 bytes	0	-32,768 to 32,767

Data Type	Size	Default Value	Range
`int`	4 bytes	0	-2^{31} to $2^{31}-1$
`long`	8 bytes	0L	-2^{63} to $2^{63}-1$
`float`	4 bytes	0.0f	6-7 decimal digits
`double`	8 bytes	0.0d	15 decimal digits
`char`	2 bytes	'\u0000'	0 to 65,535 (Unicode)
`boolean`	1 bit	false	true or false

🔹 Structure of a Java Program

Every Java program follows a well-defined structure that the JVM expects in order to compile and execute it correctly. A Java program consists of one or more **classes**, and every class is defined using the `class` keyword followed by the class name. The **main method** `public static void main(String[] args)` is the entry point of every Java application — execution always starts here. The `public` keyword makes the method accessible from outside, `static` means it can be called without creating an object, `void` means it returns nothing, and `String[] args` accepts command-line arguments. Java source files must be saved with the `.java` extension and the filename must exactly match the public class name (case-sensitive). The `javac` command compiles the source file into bytecode (`.class` file), and the `java` command runs it on the JVM. **Package declarations** go at the very top, followed by **import statements**, then the **class definition**.

```java
// Basic Java Program Structure
package com.example;           // Package declaration
import java.util.Scanner;      // Import statement

public class HelloWorld {      // Class definition
    public static void main(String[] args) {  // Main method
        System.out.println("Hello, Kashif!"); // Statement
    }
}
```

🔹 Command Line Arguments

Command Line Arguments allow users to pass data to a Java program at the time of execution, making programs more flexible and dynamic. In Java, command line arguments are received as a `String` array in the `main` method parameter `String[] args` . The first argument is `args[0]` , second is `args[1]` , and so on. To run a Java program with command line arguments, you use `java ClassName arg1 arg2 arg3` in the terminal. Since all arguments are received as Strings, they must be explicitly converted to other types using methods like `Integer.parseInt()`, `Double.parseDouble()`, etc., before performing numeric operations. The `args.length` property tells you how many arguments were passed. Command line arguments are commonly used for passing configuration values, file paths, or runtime options to programs without hardcoding them. They are particularly useful in server-side and automation applications.

```java
public class CLIDemo {
    public static void main(String[] args) {
        System.out.println("Total args: " + args.length);
        for(String arg : args) {
            System.out.println(arg);
        }
    }
}
// Run: java CLIDemo Hello Kashif 21
```

🔹 Arrays in Java

An **array** in Java is a fixed-size, ordered collection of elements of the same data type stored in contiguous memory locations. Arrays are objects in Java and are stored on the heap. A single-dimensional array is declared as `dataType[] arrayName = new dataType[size]` or initialized directly as `int[] nums = {1, 2, 3, 4, 5}`. Array elements are accessed using zero-based indexing — `arrayName[0]` gives the first element. The `.length` property (not a method) returns the number of elements. **Multi-dimensional arrays** (arrays of arrays) are used for matrices — a 2D array is declared as `int[][] matrix = new int[rows][cols]`. Java also supports **jagged arrays** where each row can have a different number of columns. Arrays are passed by reference in Java, meaning changes inside a method affect the original array. The `Arrays` class in `java.util` provides useful utility methods like `Arrays.sort()`, `Arrays.fill()`, `Arrays.copyOf()`, and `Arrays.toString()`.

🔹 Types of Arrays

Java supports several types of arrays that cater to different programming needs. A **Single Dimensional Array** is the simplest form — a linear list of elements accessed by a single index, suitable for storing simple lists like marks or names. A **Multi-Dimensional Array** (most commonly 2D) stores data in a tabular form with rows and columns — used for matrices, grids, and tables. A **Jagged Array** (also called ragged array) is a multi-dimensional array where each row has a different number of columns — useful when rows have variable lengths to save memory. An **Array of Objects** stores references to objects rather than primitive values — for example, `Student[] students = new Student[30]` creates an array of Student object references. **Anonymous Arrays** are arrays created without assigning them to a variable — often used when passing arrays directly to methods. Understanding which array type to use based on the data structure needed is an important part of efficient Java programming.

◆ String in Java

In Java, **String** is not a primitive type but a class in the `java.lang` package — it represents a sequence of characters. Strings in Java are **immutable** — once a String object is created, its value cannot be changed. Any operation that appears to modify a String actually creates a new String object. Strings can be created using string literals (`String s = "Hello"`) which are stored in the **String Pool** (a special memory area), or using `new` keyword (`String s = new String("Hello")`) which creates an object on the heap. The String class provides many useful methods — `.length()` , `.charAt()` , `.substring()` , `.indexOf()` , `.toLowerCase()` , `.toUpperCase()` , `.trim()` , `.replace()` , `.split()` , `.equals()` , and `.compareTo()` . For mutable string operations, Java provides **StringBuilder** (not thread-safe, faster) and **StringBuffer** (thread-safe, slower) classes. String comparison must always use `.equals()` method, not `==` , since `==` compares object references not content.

◆ Built-In Packages and Classes in Java

Java comes with a rich **standard library** organized into packages — pre-written classes and interfaces that provide ready-to-use functionality. The **`java.lang`** package is automatically imported in every Java program and contains fundamental classes like `String` , `Math` , `Object` , `System` , `Integer` , `Double` , `Boolean` , and `Thread` . The **`java.util`** package provides utility classes including `ArrayList` , `LinkedList` , `HashMap` , `HashSet` , `Scanner` , `Date` , `Calendar` , `Arrays` , and `Collections` . The **`java.io`** package handles input/output operations with classes like `File` , `FileReader` , `FileWriter` , `BufferedReader` , `InputStream` , and `OutputStream` . The **`java.math`** package provides `BigInteger` and `BigDecimal` for high-precision arithmetic. The **`java.net`** package supports networking with `URL` , `Socket` , and `HttpURLConnection` . To use classes from non- `java.lang` packages, you must import them using `import packageName.ClassName` or `import packageName.*` . Built-in packages dramatically reduce development time by providing tested, optimized implementations of common functionality.

🟠 Module II — Classes, Objects, Inheritance & Collections

🔹 Classes and Objects

A **Class** in Java is a blueprint or template that defines the properties (attributes) and behaviors (methods) that objects of that type will have. It is a logical entity that does not occupy memory until an object is created from it. An **Object** is a real-world instance of a class — it is a physical entity that occupies memory and has its own state and behavior. Objects are created using the `new` keyword followed by a constructor call — e.g., `Student s1 = new Student()`. Each object has its own copy of instance variables but shares class methods. A class in Java can contain **instance variables** (unique to each object), **static variables** (shared across all objects), **constructors** (special methods to initialize objects), **instance methods** (operate on object data), and **static methods** (belong to the class, not objects). The `this` keyword inside a class refers to the current object. Classes are the fundamental building blocks of any Java program and represent the encapsulation principle of OOP.

```java
class Student {
    String name;    // instance variable
    int age;

    void display() {    // instance method
        System.out.println(name + " - " + age);
    }
}

// Creating objects
Student s1 = new Student();
s1.name = "Kashif";
s1.age = 21;
s1.display();
```

🔹 Constructors in Java

A **constructor** is a special method in Java that is automatically called when an object is created using the `new` keyword. Its primary purpose is to initialize the newly created object's instance variables with valid initial values. Constructors have the same name as the class and do not have a return type — not even `void`. If no constructor is defined in a class, Java automatically provides a **Default Constructor** (no-argument constructor) that initializes all instance variables to their default values (0 for numbers, null for objects, false for booleans). A **Parameterized Constructor** accepts arguments to initialize object fields with specific values at the time of creation. **Constructor Overloading** allows a class to have multiple constructors with different parameter lists — Java determines which one to call based on the arguments provided. The `this()` call can be used inside a constructor to invoke another constructor of the same class, which is called **constructor chaining**. Constructors cannot be inherited but can be called from subclass constructors using `super()`.

◆ Method Overloading

Method Overloading is a compile-time polymorphism feature in Java that allows a class to have multiple methods with the **same name** but **different parameter lists** (different number, type, or order of parameters). The Java compiler determines which overloaded method to call based on the number and types of arguments passed at compile time — this is also called **static binding** or **early binding**. Overloading improves code readability by allowing semantically similar operations to share the same method name. The return type alone is not sufficient to differentiate overloaded methods — two methods with the same name and parameters but different return types will cause a compile-time error. **Type promotion** can occur during method overloading — if no exact match is found, Java automatically promotes smaller types to larger compatible types (e.g., `int` to `long` or `float` to `double`). Method overloading is different from method overriding — overloading happens within the same class at compile time, while overriding happens across parent-child classes at runtime.

```java
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c) { return a + b + c; }
}
```

◆ Method Overriding

Method Overriding is a runtime polymorphism feature where a subclass provides its own specific implementation of a method that is already defined in its parent class. The overriding method must have the **same name, same return type, and same parameter list** as the parent class method. The `@Override` annotation is used above the overriding method — it's optional but strongly recommended as it tells the compiler to verify that the method actually overrides a parent method, catching errors early. Method overriding enables **dynamic dispatch** — the JVM decides at runtime which version of the method to call based on the actual type of the object, not the reference type. A method declared as `final` in the parent class cannot be overridden. A `static` method cannot be overridden (it can only be hidden). The overriding method cannot have a more restrictive access modifier than the overridden method — e.g., if the parent method is `public`, the child method cannot be `protected` or `private`. The `super.methodName()` call can be used to invoke the parent class version of the overridden method.

◆ Nested Class and Inner Class

Java allows defining a class within another class — such a class is called a **Nested Class**. Nested classes logically group classes that are only used in one place, increasing encapsulation and readability. There are two types of nested classes — **Static Nested Classes** and **Inner Classes**. A **Static Nested Class** is declared with the `static` keyword and can be instantiated without an instance of the outer class — it can only access static members of the outer class. An **Inner Class** (non-static nested class) is associated with an instance of the outer class and can access all members (including private) of the outer class directly. **Local Inner Classes** are defined inside a method and can only be used within that method. **Anonymous Inner Classes** are inner classes without a name — they are declared and instantiated in a single expression, commonly used for implementing interfaces or abstract classes on the fly (e.g., event listeners). Inner classes are particularly powerful in GUI programming and implementing callbacks and event handling patterns.

◆ Overview of Inheritance

Inheritance is one of the four pillars of OOP that allows a new class (called **subclass** or **child class**) to acquire the properties and behaviors of an existing class (called **superclass** or **parent class**). Inheritance promotes **code reuse** — common attributes and methods are defined once in the parent class and automatically available in all child classes without rewriting. In Java, inheritance is implemented using the `extends` keyword — e.g., `class Dog extends Animal`. The child class inherits all **non-private** members (fields and methods) of the parent class. The child class can also add its own new fields and methods, and can override inherited methods to provide specialized behavior. Java supports **single inheritance** for classes (a class can extend only one parent class) but supports **multiple inheritance** through interfaces. The `Object` class is the root of Java's class hierarchy — every class in Java implicitly extends `Object`. Inheritance creates an **IS-A relationship** — a Dog IS-A Animal, a Car IS-A Vehicle.

◆ Types of Inheritance in Java

Java supports several forms of inheritance, each serving different design needs. **Single Inheritance** is the simplest form where one class extends exactly one parent class — e.g., `class B extends A`. **Multilevel Inheritance** creates a chain of inheritance — e.g., class C extends B, which extends A — forming a grandparent-parent-child relationship. **Hierarchical Inheritance** occurs when multiple classes extend the same single parent class — e.g., both `Dog` and `Cat` extend `Animal`. **Multiple Inheritance** (inheriting from multiple parent classes) is **NOT supported in Java for classes** to avoid the **Diamond Problem** — where ambiguity arises when two parent classes have the same method. However, Java supports multiple inheritance through **interfaces** — a class can implement multiple interfaces. **Hybrid Inheritance** is a combination of two or more types of inheritance — Java supports this only through interfaces. Understanding which inheritance type to use is crucial for designing clean, maintainable class hierarchies.

Type	Support in Java	Example
Single	✅ Yes	`class B extends A`
Multilevel	✅ Yes	`A → B → C`
Hierarchical	✅ Yes	`B extends A, C extends A`
Multiple (classes)	❌ No	Not allowed
Multiple (interfaces)	✅ Yes	`implements I1, I2`

🔹 Creation and Implementation of an Interface

An **Interface** in Java is a completely abstract type that defines a contract — a set of method signatures that implementing classes must provide. Interfaces are declared using the `interface` keyword and all their methods are implicitly `public` and `abstract` (before Java 8). Variables in interfaces are implicitly `public`, `static`, and `final` (constants). A class **implements** an interface using the `implements` keyword and must provide concrete implementations of all interface methods, or declare itself abstract. From **Java 8** onwards, interfaces can have **default methods** (with implementation, using the `default` keyword) and **static methods**. From **Java 9**, interfaces can also have **private methods**. A class can implement **multiple interfaces**, solving the multiple inheritance limitation. Interfaces define **HAS-A** or **CAN-DO** relationships — e.g., `Flyable`, `Serializable`, `Comparable`. They are fundamental to Java's design patterns, dependency injection, and writing loosely coupled, testable code.

```
interface Drawable {
    void draw();  // abstract method
    default void display() {  // default method (Java 8+)
        System.out.println("Displaying shape");
    }
}

class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
```

🔹 Abstract Class in Java

An **Abstract Class** is a class declared with the `abstract` keyword that cannot be instantiated directly — you cannot create objects of an abstract class. It serves as a base class that provides a partial implementation, leaving some methods as **abstract** (without body) to be implemented by subclasses. An abstract class can have both **abstract methods** (no implementation) and **concrete methods** (with implementation). It can also have constructors, instance variables, and static methods — unlike interfaces. A subclass must implement all abstract methods of its parent abstract class, or itself be declared abstract. Abstract classes are used when you want to provide a common base with some shared implementation while enforcing that certain behaviors must be customized by subclasses. They represent a **IS-A** relationship and are ideal for template method design patterns where the overall algorithm is defined in the abstract class but specific steps are implemented by subclasses.

🔹 Comparison Between Abstract Class and Interface

Both abstract classes and interfaces are used to achieve abstraction in Java, but they have key differences that determine when each should be used. An abstract class can have both abstract and concrete methods, while an interface (before Java 8) could only have abstract methods. An abstract class can have instance variables with any access modifier, while interface variables are always `public static final` constants. A class can extend only **one** abstract class but can implement **multiple** interfaces. Abstract classes can have constructors, interfaces cannot. Abstract classes are used when classes share a common base with partial implementation (IS-A relationship), while interfaces define capabilities or contracts that unrelated classes can implement (CAN-DO relationship). If you need to evolve your API over time, abstract classes are easier to update (add concrete methods without breaking existing code), while adding new methods to interfaces (before Java 8) would break all implementing classes.

Feature	Abstract Class	Interface
Methods	Abstract + Concrete	Abstract (+ default/static in Java 8+)
Variables	Any type	`public static final` only
Constructor	✅ Yes	❌ No
Multiple Inheritance	❌ No	✅ Yes
Keyword	`extends`	`implements`
Use Case	IS-A relationship	CAN-DO capability

🔹 Access Control in Java

Access Control in Java is implemented through **access modifiers** that determine the visibility and accessibility of classes, methods, and variables from different parts of a program. Java has four access levels — **private**, **default** (no modifier), **protected**, and **public**. `private` members are accessible only within the same class — the most restrictive level, used for data hiding and encapsulation. **Default** (package-private) members are accessible within the same package only — used when you want package-level encapsulation. `protected` members are accessible within the same package and also by subclasses in other packages — commonly used in inheritance hierarchies. `public` members are accessible from anywhere — the least restrictive level. Access control is fundamental to implementing **encapsulation** — by making fields private and providing public getter and setter methods, you control how external code reads and modifies the object's state.

◆ Creating User Defined Packages

A **package** in Java is a namespace that organizes related classes and interfaces into a group, similar to folders in a file system. Packages prevent **naming conflicts** between classes with the same name in different packages and provide **access control** — package-private members are only accessible within the package. A **user-defined package** is created by declaring `package packageName;` as the very first statement in a Java source file. The package name follows a convention of using reverse domain names — e.g., `com.kashif.college`. Classes in one package can use classes from another package by **importing** them with `import packageName.ClassName` or `import packageName.*` (imports all classes in the package). To compile a Java file with a package, use `javac -d . FileName.java` which creates the appropriate directory structure. The `java.lang` package is the only package that is automatically imported — all other packages must be explicitly imported.

◆ Java Collection Framework

The **Java Collection Framework (JCF)** is a unified architecture in `java.util` that provides ready-to-use data structures and algorithms for storing and manipulating groups of objects. It consists of **interfaces** (defining the contract), **implementations** (concrete classes), and **algorithms** (utility methods in the `Collections` class). The root interfaces are `Collection` (for single-element collections) and `Map` (for key-value pairs). The `Collection` interface is extended by `List` (ordered, allows duplicates), `Set` (unordered, no duplicates), and `Queue` (FIFO ordering). Before JCF, developers had to implement their own data structures from scratch, making Java code inconsistent and error-prone. JCF provides **interoperability** — algorithms work on any collection type through the common interfaces. The framework also includes **generics** support (since Java 5), making collections type-safe and eliminating the need for explicit casting. The `Collections` utility class provides static methods like `sort()`, `reverse()`, `shuffle()`, `min()`, `max()`, and `binarySearch()`.

🔹 Interfaces — Collection, List, Set, Iterator, List Iterator

The **Collection** interface is the root of the collection hierarchy and defines basic operations like `add()`, `remove()`, `contains()`, `size()`, `isEmpty()`, `clear()`, and `iterator()`. The **List** interface extends Collection and represents an ordered sequence that allows duplicate elements — elements can be accessed by their integer index. The **Set** interface extends Collection and represents an unordered collection that does not allow duplicate elements — attempting to add a duplicate simply returns false. The **Iterator** interface provides a standard way to traverse any collection sequentially — it has `hasNext()` (checks if more elements exist) and `next()` (returns the next element) and `remove()` (removes the last returned element). The **ListIterator** interface extends Iterator and is specific to List — it supports **bidirectional traversal** (forward and backward) with `hasPrevious()` and `previous()` methods, and also supports `add()`, `set()`, and index-based positioning using `nextIndex()` and `previousIndex()`.

🔹 Classes — LinkedList, ArrayList, Vector, HashSet

ArrayList is a resizable array implementation of the `List` interface — it provides fast random access (O(1)) but slower insertions and deletions in the middle (O(n)). It is not synchronized (not thread-safe). **LinkedList** implements both `List` and `Deque` interfaces using a doubly-linked list internally — it provides fast insertions and deletions at both ends (O(1)) but slower random access (O(n)). It is ideal for frequent add/remove operations. **Vector** is similar to ArrayList but is **synchronized** (thread-safe), making it safe for use in multi-threaded environments but slower due to synchronization overhead. Vector is considered legacy and `Collections.synchronizedList(new ArrayList<>())` is preferred in modern code. **HashSet** implements the `Set` interface using a hash table internally — it provides O(1) average performance for add, remove, and contains operations but does not maintain insertion order. HashSet allows one `null` value. **LinkedHashSet** maintains insertion order, and **TreeSet** maintains sorted order — both are alternatives to HashSet with different ordering guarantees.

Class	Interface	Order	Duplicates	Thread Safe	Performance
ArrayList	List	Insertion	✅ Yes	❌ No	Fast random access
LinkedList	List, Deque	Insertion	✅ Yes	❌ No	Fast add/remove
Vector	List	Insertion	✅ Yes	✅ Yes	Slower (synchronized)
HashSet	Set	None	❌ No	❌ No	O(1) operations

🟢 Module III — Exception Handling, File Handling & Applets

🔹 Exception and Error in Java

In Java, **exceptions** and **errors** are both subclasses of the `Throwable` class but represent fundamentally different kinds of problems. An **Exception** is an abnormal condition that occurs during program execution that can be anticipated and handled gracefully by the programmer — for example, dividing by zero, accessing a null reference, or trying to open a file that doesn't exist. An **Error** represents serious problems that are generally outside the control of the application and should not be caught — examples include `OutOfMemoryError`, `StackOverflowError`, and `VirtualMachineError`. Exceptions are further divided into **Checked Exceptions** (also called Compile-time exceptions) which must be either caught or declared in the method signature using `throws` — e.g., `IOException`, `SQLException` — and **Unchecked Exceptions** (Runtime exceptions) which do not need to be declared or caught — e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`. Understanding this hierarchy is essential for writing robust Java programs.

```
Throwable
├── Error (serious, don't catch)
│   ├── OutOfMemoryError
│   └── StackOverflowError
└── Exception (handle these)
    ├── Checked (IOException, SQLException)
    └── Unchecked/RuntimeException
        ├── NullPointerException
        ├── ArrayIndexOutOfBoundsException
        └── ArithmeticException
```

🔹 Use of try and catch

The **try-catch** block is the fundamental mechanism for handling exceptions in Java. Code that might throw an exception is placed inside the **try block**, and the code to handle the exception is placed inside the **catch block**. If an exception occurs inside the try block, the JVM immediately stops executing the try block and jumps to the matching catch block. The catch block specifies the type of exception it handles — `catch(ExceptionType e)` — and only executes if the thrown exception is of that type or a subclass. Multiple catch blocks can follow a single try block to handle different types of exceptions differently — they are evaluated top to bottom, so more specific exceptions must be caught before more general ones. From **Java 7** onwards, **multi-catch** syntax allows catching multiple exception types in a single catch block using the `|` operator — e.g., `catch(IOException | SQLException e)`. The variable `e` in the catch block represents the exception object and has useful methods like `getMessage()`, `printStackTrace()`, and `getClass().getName()`.

```java
try {
    int result = 10 / 0;  // throws ArithmeticException
    String s = null;
    s.length();           // throws NullPointerException
} catch(ArithmeticException e) {
    System.out.println("Math error: " + e.getMessage());
} catch(NullPointerException e) {
    System.out.println("Null error: " + e.getMessage());
} catch(Exception e) {   // catches any other exception
    System.out.println("General error: " + e.getMessage());
}
```

◆ throw and throws Keywords

The `throw` keyword is used to explicitly throw an exception from within a method or block of code. You use `throw` followed by an exception object — e.g., `throw new IllegalArgumentException("Age cannot be negative")`. This is useful for enforcing business rules and validations — when an invalid condition is detected, you can throw a meaningful exception to signal the problem. The `throws` keyword is used in a method signature to declare that the method might throw one or more checked exceptions — it is a warning to the calling code that it must handle these exceptions. For example, `public void readFile() throws IOException` declares that `readFile()` might throw an `IOException`. The `throws` clause does not actually throw anything — it just declares the possibility. If a method throws a checked exception without catching it, it must declare it with `throws` in its signature, or the code will not compile. You can also create **custom exceptions** by extending `Exception` (for checked) or `RuntimeException` (for unchecked) classes and throwing them with `throw`.

```java
// Custom Exception
class AgeException extends Exception {
    AgeException(String msg) { super(msg); }
}

// Method that throws it
void setAge(int age) throws AgeException {
    if(age < 0) throw new AgeException("Age cannot be negative!");
    this.age = age;
}
```

◆ finally Block

The `finally` block in Java is an optional block that follows try-catch and is guaranteed to execute regardless of whether an exception was thrown or caught. It executes even if a `return` statement is encountered in the try or catch block, and even if an exception is not caught. The finally block is primarily used for **cleanup operations** — closing database connections, releasing file handles, closing network sockets, and freeing other resources that must be released regardless of what happens. The only cases where the finally block does NOT execute are when `System.exit()` is called, when the JVM crashes, or when the thread is killed. From **Java 7** onwards, the **try-with-resources** statement is the preferred alternative for resource management — resources that implement the `AutoCloseable` interface are automatically closed at the end of the try block, making finally blocks for resource cleanup largely unnecessary. However, finally remains important for non-resource cleanup logic.

```
Connection conn = null;
try {
    conn = getConnection();
    // database operations
} catch(SQLException e) {
    System.out.println("DB Error: " + e.getMessage());
} finally {
    if(conn != null) conn.close(); // always executes
    System.out.println("Cleanup done");
}
```

🔷 Overview of Different Stream Classes in Java

Java's **I/O (Input/Output)** system is built around **streams** — a sequence of data flowing from a source to a destination. Streams are categorized in two main ways — by the **type of data** they handle and by their **direction**. **Byte Streams** handle raw binary data (8-bit bytes) and are used for all types of data including images, audio, video, and text. They are represented by the abstract classes `InputStream` and `OutputStream` at the top of the hierarchy. **Character Streams** handle text data using Unicode characters (16-bit) and are represented by abstract classes `Reader` and `Writer`. Character streams automatically handle character encoding/decoding, making them safer for text processing across different platforms. Streams can also be categorized as **Node Streams** (that directly connect to a data source/destination like files or memory) and **Filter Streams** (that wrap node streams to add functionality like buffering or data conversion). Understanding this stream hierarchy is fundamental to all I/O operations in Java.

Category	Input Class	Output Class	Use
Byte Stream	`InputStream`	`OutputStream`	Binary data
Character Stream	`Reader`	`Writer`	Text data

Category	Input Class	Output Class	Use
Buffered Byte	`BufferedInputStream`	`BufferedOutputStream`	Fast byte I/O
Buffered Char	`BufferedReader`	`BufferedWriter`	Fast text I/O
Data Stream	`DataInputStream`	`DataOutputStream`	Primitive types
Object Stream	`ObjectInputStream`	`ObjectOutputStream`	Serialization

◆ Byte Stream and Character Stream

Byte Streams process data one byte at a time and are the foundation of Java's I/O system. `FileInputStream` reads bytes from a file while `FileOutputStream` writes bytes to a file. `BufferedInputStream` and `BufferedOutputStream` wrap file streams to add an internal buffer, dramatically improving performance by reducing the number of actual disk reads/writes. `DataInputStream` and `DataOutputStream` allow reading and writing Java primitive types (int, double, boolean, etc.) in a machine-independent way. **Character Streams** process data as Unicode characters and are ideal for text file processing. `FileReader` and `FileWriter` read and write characters to files. `BufferedReader` adds buffering to character input and provides the very useful `readLine()` method that reads an entire line of text at once. `BufferedWriter` provides `newLine()` for platform-independent line endings. `InputStreamReader` and `OutputStreamWriter` are bridge classes that convert between byte streams and character streams, allowing you to specify the character encoding (UTF-8, ASCII, etc.) explicitly.

◆ Readers and Writers Class

Readers and Writers are abstract base classes for character-based I/O in Java. The `Reader` class is the superclass of all character input streams and defines the basic `read()` method. `FileReader` is the simplest concrete Reader — it reads character data from a file using the system's default character encoding. `BufferedReader` is one of the most commonly used — it wraps any Reader and adds buffering, making character reading much more efficient. Its `readLine()` method is extremely useful for reading text files line by line. `StringReader` reads characters from a String, useful for testing. `CharArrayReader` reads from a character array in memory. On the output side, `FileWriter` writes characters to a file (with option to append). `BufferedWriter` wraps any Writer with buffering and adds the `newLine()` method. `PrintWriter` is the most versatile writer — it has `print()`, `println()`, and `printf()` methods similar to `System.out`, and can wrap both streams and writers. It is commonly used for writing formatted text output to files.

◆ File Class in Java

The `File` class in `java.io` package represents a file or directory path in the file system — it does not actually read or write data but provides methods to work with file metadata and the file system itself. A `File` object is created by passing a file path string — e.g., `File f = new File("C:/data/notes.txt")`. Key methods include `exists()` (checks if the file/directory exists), `createNewFile()` (creates a new empty file), `mkdir()` and `mkdirs()` (create directories), `delete()` (deletes the file or directory), `renameTo()` (renames/moves a file), `length()` (returns file size in bytes), `getName()` (returns file name), `getPath()` (returns the path string), `isFile()` and `isDirectory()` (checks the type), `list()` (returns array of file names in a directory), and `listFiles()` (returns array of File objects in a directory). The File class is essential for file management operations like checking existence before reading, creating directory structures, and listing directory contents in Java applications.

◆ Input Stream and Output Stream

`InputStream` is the abstract superclass of all byte input stream classes in Java. It defines the basic contract for reading bytes with methods like `read()` (reads a single byte), `read(byte[])` (reads bytes into an array), `available()` (returns estimated available bytes), `skip()` (skips bytes), `mark()` and `reset()` (for repositioning in supported streams), and `close()` (releases the stream resource). `OutputStream` is the abstract superclass of all byte output stream classes. It defines `write(int)` (writes a single byte), `write(byte[])` (writes a byte array), `flush()` (forces any buffered bytes to be written), and `close()`. The most important concrete implementations include `FileInputStream` / `FileOutputStream` for file operations, `ByteArrayInputStream` / `ByteArrayOutputStream` for in-memory byte operations (useful for serialization and network transmission), `PipedInputStream` / `PipedOutputStream` for thread communication, and `ObjectInputStream` / `ObjectOutputStream` for object serialization. Always use try-with-resources or finally blocks to ensure streams are properly closed after use.

◆ Applets — Introduction

A **Java Applet** is a small Java program that runs inside a web browser or an applet viewer, embedded within an HTML page. Applets were designed in the early days of Java to bring interactivity and dynamic content to web pages — they were a revolutionary concept when introduced in 1995 as browsers were otherwise static. An applet runs in a **sandbox environment** with restricted access to the local file system and system resources for security reasons. Applets extend the `java.applet.Applet` class (AWT-based) or `javax.swing.JApplet` class (Swing-based). Unlike standalone Java applications that start with the `main()` method, applets have a lifecycle managed by the browser or applet viewer through specific methods. Applets require the **Java Plugin** to be installed in the browser. **Note:** Applets are now considered **obsolete** — major browsers dropped Java plugin support by 2015-2017, and applets were officially deprecated in Java 9 and removed in Java 17. Modern web development uses HTML5, CSS3, and JavaScript for browser-based interactivity.

◆ Types of Applets

Java applets are broadly classified into two types based on how they are created and deployed. **Local Applets** are applets that are stored and run from the **local machine** — the HTML file references the applet's `.class` file using a relative or absolute local path. Local applets do not require an internet connection and are typically used during development and testing. **Remote Applets** are stored on a **web server** and downloaded to the client's machine over the internet when the web page containing them is visited. The HTML file references the applet using a URL pointing to the server where the `.class` file is hosted. Remote applets must be digitally signed if they need to access local resources beyond the sandbox restrictions. Additionally, applets can be categorized as **AWT Applets** (using the older Abstract Window Toolkit for UI) and **Swing Applets** (using the more modern Swing library that extends `JApplet` and provides a richer, more consistent UI across platforms).

◆ Applet Lifecycle

The **Applet Lifecycle** is controlled by the browser or applet viewer through four key methods that are automatically called at specific points during the applet's existence. Understanding these lifecycle methods is essential for writing correct applets that properly initialize, run, pause, resume, and clean up resources.

`init()` — Called once when the applet is first loaded into memory. Used for one-time initialization tasks like setting up UI components, loading resources, and initializing variables. Equivalent to a constructor for applets.

`start()` — Called after `init()` completes, and also called every time the user returns to the page containing the applet (after navigating away). Used to start or resume animations, threads, or any ongoing activity.

`stop()` — Called when the user navigates away from the page or minimizes the browser. Used to pause animations, suspend threads, and stop resource-intensive operations. The applet remains in memory.

`destroy()` — Called when the applet is about to be permanently removed from memory (browser closing or page refresh). Used for final cleanup — releasing all resources, closing connections, and stopping all threads.

`paint(Graphics g)` — Called whenever the applet needs to be redrawn — when first displayed, when the window is resized, or when `repaint()` is called programmatically. All drawing and rendering happens here.

```
load → init() → start() → paint()
                   ↑             ↓
              (return)     stop() → (navigate away)
                              ↓
                           destroy() → (browser closes)
```

🔹 Creating an Applet

Creating a Java applet involves extending the `Applet` class and overriding the necessary lifecycle methods. The applet is then compiled and embedded in an HTML file using the `<applet>` tag (deprecated) or `<object>` tag. A basic applet overrides the `paint()` method which receives a `Graphics` object that provides methods for drawing shapes, text, and images on the applet's display area.

```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class WelcomeApplet extends Applet {
    public void init() {
        setBackground(Color.lightGray);
    }

    public void paint(Graphics g) {
        g.setColor(Color.blue);
        g.setFont(new Font("Arial", Font.BOLD, 20));
        g.drawString("Welcome, Kashif Sayyad!", 50, 100);
        g.setColor(Color.red);
        g.drawRect(30, 70, 300, 50);
    }
}
```

```html
<!-- HTML to embed applet -->
<applet code="WelcomeApplet.class" width="400" height="200">
</applet>
```

🔹 Applet Tag and Applet Examples

The `<applet>` HTML tag was the original way to embed Java applets in web pages, though it is now obsolete and replaced by the `<object>` tag. The `<applet>` tag has several important attributes — `code` specifies the `.class` file of the main applet class, `width` and `height` define the applet's display dimensions in pixels, `codebase` specifies the directory containing the applet's class files (if different from the HTML file's location), `archive` specifies a JAR file containing the applet's classes, `name` gives the applet an identifier for scripting, and `alt` provides text for browsers that don't support applets. Content between the opening and closing `<applet>` tags is displayed as fallback text for browsers that cannot run applets. **Parameters** can be passed to applets using `<param name="paramName" value="paramValue">` tags inside the applet tag, and retrieved inside the applet using `getParameter("paramName")`.

◆ Passing Parameters to an Applet

Passing parameters to applets allows the same applet code to behave differently based on the values specified in the HTML, making applets reusable and configurable. Parameters are defined in the HTML using `<param>` tags nested inside the `<applet>` tag, and retrieved inside the applet using the `getParameter(String name)` method which returns the parameter value as a String (or null if the parameter is not found). Since all parameters are returned as Strings, numeric values must be converted using `Integer.parseInt()`, `Double.parseDouble()`, etc. Parameters should always be validated and default values should be provided in case `getParameter()` returns null. Parameters are typically read in the `init()` method since they need to be available before the applet starts displaying content. This mechanism is analogous to passing command-line arguments to a standalone Java application, enabling the HTML author to customize the applet's behavior without modifying the Java source code.

```java
public class ParamApplet extends Applet {
    String userName;
    int fontSize;

    public void init() {
        userName = getParameter("name");
        if(userName == null) userName = "Guest";

        String size = getParameter("size");
        fontSize = (size != null) ? Integer.parseInt(size) : 16;
    }

    public void paint(Graphics g) {
        g.setFont(new Font("Arial", Font.BOLD, fontSize));
        g.drawString("Hello, " + userName + "!", 50, 80);
    }
}
```

```html
<applet code="ParamApplet.class" width="400" height="150">
    <param name="name" value="Kashif Sayyad">
    <param name="size" value="24">
</applet>
```
