JAVA UNIT 3

1. Abstract Class and Abstract Methods

- An **abstract class** is a class that is declared using the abstract keyword and contains **partial implementation**. This means some methods may have a body, while others may not have any implementation.
- An **abstract method** is a method that has only a declaration and **no method body**. It is declared using the abstract keyword and ends with a semicolon.
- The **purpose of an abstract class** is to provide a common structure for subclasses. It defines what methods subclasses must implement, without forcing a fixed implementation.
- Abstract classes **cannot be instantiated**, but their references can be created to support runtime polymorphism.
- **Rules**:
  Any class containing at least one abstract method must be declared abstract.
  Subclasses must implement all abstract methods or themselves be abstract.
- **Syntax**:
  abstract class ClassName { abstract returnType methodName(); }
- **Example**:
  A Bike abstract class defines run() as abstract. The Honda class extends it and provides the implementation, ensuring consistent behavior across subclasses.

2. Interface: Definition, Syntax, Purpose, Rules & Program

- An **interface** in Java is a blueprint of a class that contains **only abstract methods and constants**. It defines what a class must do, but not how it does it.
- The **purpose of an interface** is to achieve **full abstraction** and support **multiple inheritance**, which is not possible using classes alone.
- Interfaces help in creating **loose coupling**, where classes depend on behavior rather than specific implementations.
- **Syntax**:
  interface InterfaceName { returnType methodName(); }
- **Rules**:
  All methods are implicitly public and abstract. All variables are public, static, and final. Interfaces cannot be instantiated.
- **Allowed members**:
  Interfaces allow only constants (static final variables) and abstract methods. From Java 8, default and static methods are allowed.
- **Program explanation**:
  In the PPT example, the Animal interface is implemented by the Dog class. The interface reference points to a Dog object, showing abstraction and runtime polymorphism.

3. Difference Between Abstract Class and Interface & Use Scenarios

- An **abstract class** can have both abstract and non-abstract methods, while an **interface** mostly contains abstract methods only.
- Abstract classes **do not support multiple inheritance**, whereas interfaces support multiple inheritance by allowing a class to implement multiple interfaces.
- Abstract classes can have **instance variables**, but interfaces can only have static final variables.
- Abstract classes are extended using the extends keyword, while interfaces are implemented using the implements keyword.
- **Use abstract class** when classes are closely related and share common code. It is useful when partial implementation is required.
- **Use interface** when unrelated classes need to follow a common contract. It is ideal when multiple inheritance and full abstraction are required.
- Thus, abstract classes focus on **code reuse**, while interfaces focus on **standardization and flexibility**.

4. Abstract Class Shape with Circle and Rectangle

- The abstract class Shape is designed to represent common behavior for different shapes. It contains an abstract method calculateArea().
- The method calculateArea() is abstract because the formula for area differs for each shape. This forces subclasses to provide their own implementation.
- The Circle class extends Shape and calculates area using the formula π × r × r. This implementation is specific to circles.
- The Rectangle class also extends Shape and calculates area using length × breadth. This shows how different subclasses handle the same method differently.
- The **implementation process** starts by declaring an abstract class, then extending it using subclasses. Each subclass overrides the abstract method.
- This design improves **flexibility, reusability, and abstraction**, ensuring that all shapes follow the same structure while allowing different logic.

5. Runtime Polymorphism Using Abstract Class

- **Runtime polymorphism** occurs when a superclass reference refers to a subclass object. The method call is resolved at runtime based on the object type.
- Abstract classes support runtime polymorphism by allowing **references to abstract classes**, even though objects cannot be created from them.
- In the PPT example, the abstract class Figure defines an abstract method area(). Subclasses like Rectangle and Triangle implement it differently.
- A Figure reference is used to point to different subclass objects. When area() is called, the corresponding subclass method executes.
- This mechanism allows the same method call to produce different results, depending on the object type.
- Runtime polymorphism increases **flexibility, extensibility, and code maintainability**, making programs easier to modify and extend without changing existing code.

6. Runtime Polymorphism Using Interface

- **Runtime polymorphism** using interface is achieved when an interface reference variable refers to an object of a class that implements the interface. The method call is resolved at runtime based on the object type.
- Interfaces allow multiple classes to provide their own implementation of the same method. This helps in achieving dynamic method resolution.
- The interface defines the method structure, while the implementing class provides the actual logic. This separates *what to do* from *how to do*.
- In the example from the PPT, an interface FigureI is implemented by classes CircleI and SquareI. Each class provides its own version of area() and perimeter() methods.
- A reference of type FigureI is used to point to different objects at different times. The correct method is called based on the object created at runtime.
- This approach improves flexibility, scalability, and code maintainability by allowing behavior changes without modifying existing code.

7. Multiple Inheritance in Java Using Interface

- **Multiple inheritance** means a class inheriting properties from more than one parent. Java does not support multiple inheritance using classes due to ambiguity issues.
- Java supports multiple inheritance using **interfaces**, because interfaces contain only method declarations and no implementation conflicts.
- A class can implement more than one interface using the implements keyword. This allows the class to inherit behavior from multiple sources.
- Interfaces themselves can also extend multiple interfaces. This further enhances reusability and flexibility.
- In the PPT example, a class implements both Printable and Showable interfaces. It provides implementations for methods from both interfaces.
- This mechanism avoids ambiguity and maintains clear method definitions. Hence, Java safely supports multiple inheritance through interfaces only.

8. Packages: Definition, Purpose, Types & Examples

- A **package** in Java is a mechanism used to group related classes and interfaces together. It helps in organizing large programs into logical units.
- The **purpose of a package** is to avoid class name conflicts and manage namespace efficiently. It also provides access control and encapsulation.
- Packages improve code reusability and make large applications easier to maintain. They also enhance security by restricting access to certain classes.
- **Types of packages** are:
    - Built-in packages
    - User-defined packages
- **Examples of built-in packages**:

1. java.lang
2. java.util
3. java.io
4. java.net
5. java.awt

- Built-in packages provide predefined classes and interfaces, while user-defined packages are created by programmers based on application needs.

9. Difference Between Built-in and User-Defined Packages

- **Built-in packages** are provided by Java as part of the Java API. They contain predefined classes that perform common tasks.
- Examples of built-in packages include java.lang, java.util, and java.io. These packages are readily available and require no creation.
- **User-defined packages** are created by programmers to organize their own classes. They help manage large projects efficiently.
- Built-in packages are automatically available or imported, while user-defined packages must be explicitly created and imported.
- Built-in packages support standard functionality, whereas user-defined packages support application-specific logic.
- Both types of packages improve modularity and readability, but user-defined packages give developers more control over code organization.

10. Steps to Create and Import a User-Defined Package

- First, create a Java file and write the package statement as the **first line** of the program. This defines the package name.
- Write the class inside the package and save the file. The directory structure should match the package name.
- Compile the package file using the Java compiler. The folder structure will be created automatically if not present.
- To use the package in another program, use the import packageName.*; or import packageName.ClassName; statement.
- Alternatively, the **fully qualified name** can be used while creating the object.
- The program sequence must always follow: **package → import → class**. This ensures correct compilation and execution.

11. Access Modifiers: private, public, protected, package-private

- **Access modifiers** in Java control the visibility of classes, variables, and methods. They help in implementing data hiding and security.
- **Public** members are accessible from anywhere in the program. If a class or method is declared public, it can be accessed from any package.
- **Private** members are accessible only within the same class. They are mainly used to protect sensitive data from direct access.
- **Protected** members are accessible within the same package and also in subclasses outside the package. This is useful when inheritance is involved.

- **Package-private (default)** access is applied when no access modifier is specified. Such members are accessible only within the same package.
- **Example explanation**:
  In the PPT example, variables with different access modifiers are declared in class A1. When accessed from another package, private and default members are not visible, while public and protected members behave according to rules.
- Access modifiers ensure **controlled access**, **security**, and **better program structure**.

12. Exception and Exception Handling

- An **exception** is an abnormal condition that occurs during the execution of a program. It disrupts the normal flow of instructions.
- Examples of exceptions include division by zero, array index out of bounds, and null pointer access.
- **Exception handling** is a mechanism used to handle runtime errors gracefully. It prevents abrupt termination of the program.
- The **purpose of exception handling** is to maintain normal program flow even when errors occur. It also helps in displaying meaningful error messages.
- Java provides keywords like try, catch, finally, throw, and throws to handle exceptions.
- By handling exceptions, programmers can separate error-handling code from normal logic. This improves readability and reliability.
- Exception handling also helps in debugging and recovering from errors, making applications more robust and user-friendly.

13. Difference Between Checked and Unchecked Exceptions

- **Checked exceptions** are exceptions that are checked at compile time. The compiler forces the programmer to handle or declare them.
- Examples of checked exceptions include IOException, SQLException, and FileNotFoundException.
- If a checked exception is not handled using try-catch or declared using throws, the program will not compile.
- **Unchecked exceptions** are not checked at compile time. They occur during runtime due to programming errors.
- Examples of unchecked exceptions include ArithmeticException, NullPointerException, and ArrayIndexOutOfBoundsException.
- Unchecked exceptions inherit from the RuntimeException class. They are usually caused by logical mistakes.
- In short, checked exceptions ensure safety at compile time, while unchecked exceptions indicate runtime errors.

14. Exception Class Hierarchy

- In Java, all exceptions are subclasses of the **Throwable** class. It is the root of the exception hierarchy.
- The Throwable class has two main subclasses: **Error** and **Exception**.
- **Error** represents serious problems that applications should not try to handle. Examples include StackOverflowError.
- **Exception** represents conditions that programs can catch and handle. It includes both checked and unchecked exceptions.
- The RuntimeException class is a subclass of Exception. It represents unchecked exceptions.
- Common runtime exceptions include ArithmeticException and NullPointerException.
- This hierarchy helps Java distinguish between recoverable and non-recoverable errors, allowing structured and effective exception handling.

15. try, catch, and finally Blocks

- The **try block** contains code that may cause an exception. It is always enclosed within curly braces.
- The **catch block** handles the exception thrown by the try block. Each catch block handles a specific type of exception.
- Multiple catch blocks can be used after a try block. The most specific exception must be caught first.
- The **finally block** contains code that is always executed, whether an exception occurs or not. It is mainly used for resource cleanup.
- **Purpose**:
  Try prevents abnormal termination, catch handles errors, and finally ensures execution of important code.
- **Syntax**:
  try {

  } catch(Exception e) {

  } finally {

  }
- **Example explanation**:
  In the PPT example, division by zero is handled using a catch block, and the program continues execution normally after handling the exception.

16. Purpose of throw and throws Statements

- The **throw statement** is used to explicitly throw an exception from a program. It allows the programmer to create and generate an exception manually.
- Using throw, a programmer can signal an abnormal condition intentionally. This helps in enforcing program rules and validations.
- The syntax of throw is:
  throw new ExceptionType("message");
- The **throws statement** is used in a method declaration to indicate that the method may pass an exception to the calling method. It does not handle the exception itself.
- throws is mainly used with **checked exceptions**. It informs the compiler and caller about possible exceptions.
- Syntax of throws:
  returnType methodName() throws ExceptionType
- **Example explanation**:
  In the PPT example, an exception is thrown using throw, and the same exception is re-thrown to the calling method where it is handled.
- In short, throw creates an exception, while throws declares it.

17. User-Defined Exception (InvalidAgeException)

- A **user-defined exception** is a custom exception created by the programmer. It is created by extending the Exception class.
- User-defined exceptions help handle application-specific errors clearly. They improve program readability and control.
- In the PPT example, InvalidAgeException is created to check voting eligibility. This exception is thrown when age is below 18.
- The custom exception class contains a constructor that passes the error message to the parent Exception class.
- The method validate() checks the age and throws InvalidAgeException if the condition fails. This enforces business rules effectively.
- The exception is handled using a try-catch block in the main method. This prevents program termination.
- User-defined exceptions make error handling meaningful and allow developers to define their own validation logic.

18. Checked Exception (Purpose, Syntax, Rules, Example)

- **Checked exceptions** are exceptions that are checked at **compile time**. The compiler forces the programmer to handle or declare them.
- The main purpose of checked exceptions is to ensure safer code execution. They prevent runtime failures by handling errors early.
- Checked exceptions inherit from the Exception class but not from RuntimeException.
- **Rules**:
  A method that throws a checked exception must either handle it using try-catch or declare it using throws.
- **Syntax**:
  methodName() throws IOException
- **Example explanation**:
  In the PPT example, FileReader and readLine() throw IOException. If not handled, the program will not compile.
- The corrected version uses throws IOException in the method declaration. This allows the program to compile and run.
- Checked exceptions improve reliability by forcing proper exception handling.

19. Unchecked Exception (Purpose, Syntax, Rules, Example)

- **Unchecked exceptions** are exceptions that are **not checked at compile time**. They occur during program execution.
- These exceptions usually arise due to programming mistakes. Common causes include division by zero and invalid array index access.
- Unchecked exceptions inherit from the RuntimeException class. The compiler does not force handling of these exceptions.
- **Purpose**:
  They indicate logical errors in the program. Fixing the code logic usually resolves them.
- **Syntax**:
  No special syntax is required. They occur automatically during runtime.
- **Example explanation**:
  In the PPT example, dividing a number by zero causes ArithmeticException. The program compiles successfully but crashes at runtime.
- **Rules**:
  Handling unchecked exceptions is optional. However, proper validation can prevent them.
- Unchecked exceptions highlight runtime issues and help in debugging faulty logic.