Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto- unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java.time.Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions.

Java I/O and File: Java I/O API, standard I/O streams, types, Byte streams, Character streams, Scanner class, Files in Java

1. Introduction to Packages in Java

In Java, a **package** is a mechanism for organizing Java classes, interfaces, and sub-packages into namespaces. Packages act like containers that group related classes and interfaces, helping to avoid naming conflicts and managing large codebases efficiently.

Key Benefits of Using Packages:

1. **Namespace Management**: Packages help in organizing classes and interfaces into different namespaces, which prevents naming conflicts. For example, you can have two classes with the same name in different packages without causing any conflicts.
2. **Access Control**: Packages allow the application of access control. Classes, methods, and fields can be declared `public`, `protected`, `private`, or package-private (default), controlling how they are accessed from other packages or within the same package.
3. **Code Reusability**: Packages make it easier to reuse classes across different projects or parts of a project. You can easily import them into other programs and extend their functionality.
4. **Logical Grouping**: Grouping related classes together makes it easier to maintain and manage code. It also provides structure, making the code more readable and understandable.

package:

- Package a group of similar types of classes and interfaces and subpackages

Or

- Package is a folder that contains collection of related classes and interfaces.
- In java ,packages can be categorized into two types
- 1.Built-in packages

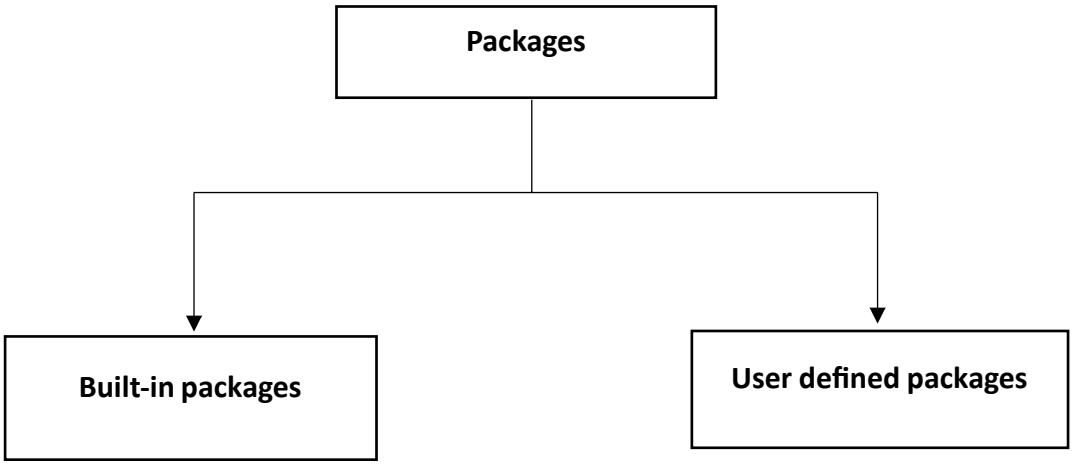
  2.user-defined packages

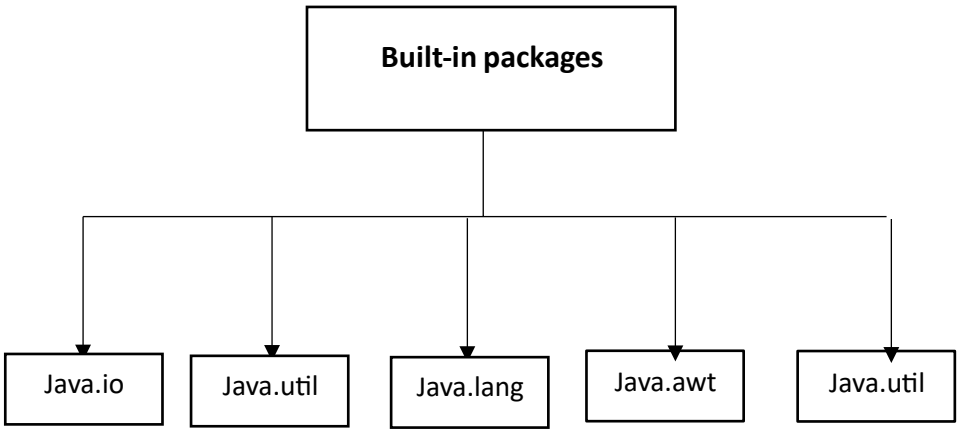
Built-in packages

In java we have various built-in packages that are already created by java people and these packages contain large number of classes and interfaces

User defined packages

As the name suggests user-defined packages are a package that is defined by theuser or programmer.





Java.io	Contains classes related to input/output operations
Java.util	Contains classes and interfaces of collection framework, scanner class
Java.lang	Contains fundamental classes like system, object etc for designing java program
Java.awt	Contains classes and interfaces for creating graphical components
Java.swing	Contain classes and interfaces for creating graphical components

Advantages

- Java package is used to categorized the classes and interfaces so that theycan be easily categorized.
- Java package provides access protection
- Java package helps to avoid name space collision.

How to create user defined packages

- To create the package should be starts with the keyword is **package**

  Syntax: package package_name;
- It should not contain main class
- Multiple programs should be written for placing multiple classes in samepackage.

Steps to create a simple user defined packageStep-1

```
package pack;
public class PackDemo
{
        public void show()
        {
                System.out.println("welcome to java");
        }
}
```

Step-2

```
import pack.PackDemo;
class Pack1
{
        public static void main(String args[])
        {
                PackDemo obj=new PackDemo();
                obj.show();
        }
}
```

Step-3

```
D:\java>javac -d . PackDemo.java

D:\java>
```

Step-4

```
D:\java>javac Pack1.java

D:\java>java Pack1
welcome to java

D:\java>
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package java.awt.image;

Example: Package demonstration

```
package   pack; public
class Addition
{
        int x,y;
        public Addition(int a, int b)
        {
                x=a;
                y=b;
        }
        public void sum()
        {
                System.out.println("Sum :"+(x+y));
```

```
        }
}
```

Step 1: Save the above file with Addition.java

```
package pack;
public class Subtraction
{
        int x,y;
        public Subtraction(int a, int b)
        {
                x=a;
                y=b;
         }
        public void diff()
        {
                System.out.println("Difference :"+(x-y));
        }
}
```

Step 2: Save the above file with Subtraction.javaStep 3:

Compilation

To compile the java files use the following commands

javac -d directory_path name_of_the_java fileJavac –d

. name_of_the_java file

Note: -d is a switching options creates a new directory with package name. Directory path represents in which location you want to create package and . (dot)represents current working directory.

Step 4: Access package from another package

There are three ways to use package in another package:

1. **With fully qualified name.**

```
class UseofPack


{
    public static void main(String arg[])
    {
        pack.Addition a=new pack.Addition(10,15);
        a.sum();
        pack.Subtraction s=new pack.Subtraction(20,15);
        s.difference();
    }
}
```

2. **import package.classname;**

```
import pack.Addition;
import pack.Subtraction;
class UseofPack
{
    public static void main(String arg[])
    {
        Addition a=new Addition(10,15);a.sum();
        Subtraction s=new Subtraction(20,15);
        s.difference();
    }
}
```

3. **import package.*;**

```
import pack.*; class
UseofPack
   {
       public static void main(String arg[])
      {
          Addition a=new Addition(10,15);a.sum();
          Subtraction s=new Subtraction(20,15);
          s.difference();
      }
    }
```

Note: Don't place Addition.java, Subtraction.java files parallel to the pack directory. If you place JVM searches for the class files in the current working directory not in the pack directory.

Access Control

- Java provides four types of **access modifiers**: `public`, `protected`, `default` (no modifier), and `private`.
    - `public`: Accessible from any class.
    - `protected`: Accessible within the same package and subclasses.
    - `default`: Accessible only within the same package.
    - `private`: Accessible only within the class where it is declared.

Packages in Java SE

1. java.lang Package

- This package is automatically imported into every Java program, providing fundamental classes essential for the language.
- **Key classes:**
    - Object: The root class from which all classes in Java inherit.
    - String: Immutable sequences of characters.
    - Math: Provides mathematical operations such as sqrt(), pow(), abs().
    - System: Used to interact with system resources, e.g., System.out for standard output.
    - Thread: For multithreading operations.

2. java.util Package

- Contains utility classes and interfaces used for collections, date/time manipulation, and random number generation.

- Key classes:
  - ArrayList, LinkedList, HashSet, HashMap: For handling dynamic collections of data.
  - Collections: Utility class for manipulating collections (e.g., sorting, searching).
  - Date, Calendar, TimeZone: For handling date and time.
  - Random: For generating random numbers.

3. java.io Package

- Provides classes for input and output operations, such as reading and writing data to files, handling streams, and working with serializable objects.
- Key classes:
  - File: Represents file and directory pathnames.
  - BufferedReader, BufferedWriter: For efficient reading/writing of text from/to files.
  - InputStream, OutputStream: Base classes for byte stream operations.
  - Serializable: Marks classes for object serialization

Wrapper Classes in Java

Wrapper classes in Java are used to convert **primitive data types** into **objects**. Each of Java's eight primitive types (int, char, etc.) has a corresponding wrapper class in the `java.lang` package. These wrapper classes provide a way to treat primitive data types as objects, which is necessary in scenarios where only objects are allowed, such as with Java Collections (e.g., `ArrayList`, `HashMap`).

The process of converting a primitive type to its corresponding wrapper object is known as **boxing**, and converting it back to a primitive is called **unboxing**.

Primitive Types and Corresponding Wrapper Classes:

Primitive Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`

Primitive Type	Wrapper Class
short	Short
int	Integer
long	Long
float	Float
double	Double

Why Use Wrapper Classes?

1. **Object-Oriented Collection Classes**: Java's collection classes (e.g., `ArrayList`, `HashMap`) can only store objects, not primitives. Wrapper classes allow primitive data types to be stored in collections by converting them into objects.
2. **Utility Methods**: Wrapper classes provide many useful methods for manipulating and converting primitive values.
3. **Default Values in Generics**: Java Generics work only with objects, so wrapper classes are used when you need to work with generic types.
4. **Nullability**: Wrapper classes can be `null`, whereas primitive types cannot. This can be useful for representing the absence of a value.

Boxing and Unboxing

- **Boxing** is the process of converting a primitive type into its corresponding wrapper object.
- **Unboxing** is the reverse process, where the wrapper object is converted back into a primitive type.

Example of Boxing and Unboxing:

```
public class BoxingUnboxingExample {

   public static void main(String[] args) {

      // Boxing (primitive to object)

      int num = 100;

      Integer obj = Integer.valueOf(num); // explicitly boxing


      // Unboxing (object to primitive)

      Integer obj1 = new Integer(200);

      int num2 = obj1.intValue(); // explicitly unboxing
```

```
        System.out.println("Boxed Integer: " + obj);
        System.out.println("Unboxed int: " + num2);
    }
}
```

Auto Boxing and Auto Unboxing

Java automatically handles the conversion between primitives and their corresponding wrapper classes through **auto-boxing** and **auto-unboxing**.

- **Auto-boxing**: Automatic conversion of a primitive type into its wrapper class object.
- **Auto-unboxing**: Automatic conversion of a wrapper object to its corresponding primitive type.

Example of Auto Boxing and Auto Unboxing:

```
public class AutoBoxingUnboxingExample {
    public static void main(String[] args) {
        // Auto-boxing
        int num = 100;
        Integer obj = num; // no need to call Integer.valueOf(num)

        // Auto-unboxing
        Integer obj1 = new Integer(200);
        int num2 = obj1; // no need to call wrappedNum2.intValue()

        System.out.println("Auto-boxed Integer: " + obj);
        System.out.println("Auto-unboxed int: " + num2);
    }
}
```

Java util Classes and Interfaces

Formatter Class (java.util.Formatter)

The Formatter class in Java provides support for formatting data (such as strings, numbers, dates, etc.) in a way similar to printf() in C. It can format output based on a format string that specifies how data should be presented. It is often used in logging, console output, or file writing.

Key Methods:

- **format()**: This is the primary method for formatting. It supports a variety of data types, and the format string uses placeholders.

Key Concepts:

1. **Format String**: The format string specifies how data should be formatted. It contains placeholders like %d, %f, %s, which get replaced with actual values.
2. **Supported Data Types**:
    - **%d**: Integer (decimal).
    - **%f**: Floating-point number (decimal).
    - **%s**: String.
    - **%x**: Integer (hexadecimal).
    - **%o**: Integer (octal).
    - **%t**: Date/time values.

Example:

```
public class FormatterExample {
   public static void main(String[] args) {
      Formatter fmt = new Formatter();
      fmt.format("Value of Pi to 2 decimals: %.2f", 3.14159);
      System.out.println(fmt);
      fmt.close();
   }
}
```

Output:

```
Value of Pi to 2 decimals: 3.14
```

Example2:

```
import java.util.Formatter;

public class FormatterExample {
   public static void main(String[] args) {
      // Create a Formatter
      Formatter fmt = new Formatter();

      // Format an integer, a float, and a string
      fmt.format("Integer: %d\n", 123);
      fmt.format("Floating-point: %.2f\n", 3.14159);
      fmt.format("String: %s\n", "Hello, World!");
```

```
        // Print the formatted output
        System.out.println(fmt);

        // Close the Formatter to release resources
        fmt.close();
    }
}
```

Output:

Integer: 123

Floating-point: 3.14

String: Hello, World!

Formatting Dates and Times:

You can use the Formatter class to format dates and times using the %t prefix.

- **%tY**: Year (4 digits).
- **%tm**: Month (2 digits).
- **%td**: Day of the month.
- **%tH**: Hour (24-hour clock).
- **%tM**: Minute.
- **%tS**: Second.

Example3:

```
import java.util.Formatter;
import java.util.Calendar;

public class DateFormatExample {
    public static void main(String[] args) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        // Format current date and time
        fmt.format("Current Date: %tY-%tm-%td\n", cal, cal, cal);
        fmt.format("Current Time: %tH:%tM:%tS\n", cal, cal, cal);

        // Print the formatted output
        System.out.println(fmt);
```

```
        // Close the Formatter
        fmt.close();
    }
}
```

Output:

Current Date: 2024-10-13

Current Time: 09:30:47

2. Random Class (java.util.Random)

The Random class in Java is used to generate pseudo-random numbers. It provides methods to generate random integers, floats, longs, and even boolean values.

Key Methods:

- **nextInt()**: Returns a random integer.
- **nextInt(int bound)**: Returns a random integer within the specified bound.
- **nextDouble()**: Returns a random double between 0.0 and 1.0.
- **nextBoolean()**: Returns a random boolean.

Example:

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        Random random = new Random();

        // Generate random integers
        int randInt = random.nextInt(100);  // Random integer between 0 and 99
        System.out.println("Random Integer: " + randInt);

        // Generate random doubles
        double randDouble = random.nextDouble();  // Random double between 0.0 and 1.0
        System.out.println("Random Double: " + randDouble);

        // Generate random booleans
        boolean randBoolean = random.nextBoolean();
```

```
        System.out.println("Random Boolean: " + randBoolean);
    }
}
```

Output:

Random Integer: 70

Random Double: 0.024016527282495925

Random Boolean: false

3. Time Package (java.time)

The java.time package introduced in Java 8 provides a comprehensive API for handling dates and times. It offers a much more flexible and modern way of working with time compared to the legacy java.util.Date and java.util.Calendar classes.

Key classes include:

- **LocalDate**: Represents a date without time.
- **LocalTime**: Represents a time without a date.
- **LocalDateTime**: Represents a date and time.
- **Duration**: Represents a time duration (e.g., 5 hours, 30 minutes).
- **Period**: Represents a date-based amount of time (e.g., 2 years, 3 months).
- **ZonedDateTime**: Represents a date-time with a time zone.

Example:

1. LocalDate

Represents a date without a time zone (year, month, day).

```
import java.time.LocalDate;

public class LocalDateExample {
    public static void main(String[] args) {
        // Get the current date
        LocalDate today = LocalDate.now();
        System.out.println("Today's date: " + today);

        // Create a specific date
```

```
        LocalDate specificDate = LocalDate.of(2024, 10, 13);
        System.out.println("Specific date: " + specificDate);

        // Add days to a date
        LocalDate nextWeek = today.plusDays(7);
        System.out.println("Date after one week: " + nextWeek);

        // Check if a year is a leap year
        boolean isLeapYear = today.isLeapYear();
        System.out.println("Is this year a leap year? " + isLeapYear);
    }
}
```

Output:

```
Today's date: 2024-10-13
Specific date: 2024-10-13
Date after one week: 2024-10-20
Is this year a leap year? false
```

Example2:

2. LocalTime

Represents a time without a date and without a time zone.

```
import java.time.LocalTime;

public class LocalTimeExample {
    public static void main(String[] args) {
        // Get the current time
        LocalTime now = LocalTime.now();
        System.out.println("Current time: " + now);

        // Create a specific time
        LocalTime specificTime = LocalTime.of(14, 30, 45); // 2:30:45 PM
        System.out.println("Specific time: " + specificTime);
```

```
        // Add hours and minutes to the current time
        LocalTime later = now.plusHours(2).plusMinutes(15);
        System.out.println("Time after 2 hours and 15 minutes: " + later);
        // Get the hour, minute, and second
        int hour = now.getHour();
        int minute = now.getMinute();
        int second = now.getSecond();

        System.out.println("Hour: " + hour + ", Minute: " + minute + ", Second: " + second);
    }
}
```

Output:

Current time: 09:30:47.123

Specific time: 14:30:45

Time after 2 hours and 15 minutes: 11:45:47.123

Hour: 9, Minute: 30, Second: 47

5. Formatting for Date/Time in Java (DateTimeFormatter)

The DateTimeFormatter class (from java.time.format) is used to format and parse date/time objects. It provides flexible and powerful formatting options.

Common Predefined Formatters:

- **ISO_LOCAL_DATE**: Formats a date as yyyy-MM-dd.
- **ISO_LOCAL_DATE_TIME**: Formats a date and time as yyyy-MM-ddTHH:mm:ss.

Custom Format Example:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeFormattingExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
```

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

        System.out.println("Formatted DateTime: " + formattedDateTime);
    }
}
```

Output:

Formatted DateTime: 13-10-2024 08:52:35

6. TemporalAdjusters Class (java.time.temporal.TemporalAdjusters)

The TemporalAdjusters class provides common temporal adjusters, which allow date manipulations such as finding the next day of the week, the last day of the month, etc. Adjusters are often used with classes like LocalDate.

Common Temporal Adjusters:

- **firstDayOfMonth()**: Returns the first day of the current month.
- **lastDayOfMonth()**: Returns the last day of the current month.
- **next(DayOfWeek dayOfWeek)**: Returns the next occurrence of the specified day of the week.
- **previous(DayOfWeek dayOfWeek)**: Returns the previous occurrence of the specified day of the week.

Example:

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class TemporalAdjustersExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        // Get the next Sunday
        LocalDate nextSunday = today.with(TemporalAdjusters.next(DayOfWeek.SUNDAY));
        System.out.println("Next Sunday: " + nextSunday);
```

```
        // Get the last day of the current month

        LocalDate lastDayOfMonth = today.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("Last Day of Month: " + lastDayOfMonth);
    }
}
```

Output:

Next Sunday: 2024-10-20

Last Day of Month: 2024-10-31

Exception Handling

Java Errors are classified into 3 types

1) Compile -Time Errors
2) Run -Time Errors
3) Logical Errors

Compile-Time Errors

- Errors occurred at Compile Time are called Compile Time Errors
- Syntax Errors are detected at Compile Time.
- **These are Syntactical Errors found in the code, due to which a program fails to compile.**
- For Example, forgetting a semicolon at the end of a Java statement, or writing a statement without proper syntax will result in compile-time error.
- Detecting and Correcting compile-time errors is easy as the Java Compiler displays the list of errors with the line numbers along with theirdescription

Run Time Errors

- Errors occurred at **Run Time** are called Run Time Errors
- Run time errors are not detected by the **java compiler**.
- It is the JVM which detects it while the program is running.
- **Semantic Errors** like division by zero, Index out of Bound are detectedby JVM at runtime.

Logical Errors

- These errors are due to the mistakes made by the programmer.
- It will not be detected by a **compiler nor by the JVM.**
- errors may be due to wrong idea or concept used by a programmer while coding.

Introduction to Exception Handling

- An Exception is a **Run Time Error** (or) An exception is **abnormal condition** that arises in a code sequence at the **run time.**
- When the jvm encounter an **Run Time Error** such as **Division by zero**, JVM creates an object to the Corresponding Class and throws it.
- If the Programmer does not catch the thrown object and handles properly, ***the interpreter will display an error message and the program gets terminated abnormally***.
- **In order to stop abnormal termination of the program and to fix the error, exceptions should be caught and handled.**

Java exception handling is managed via five keywords

- Try
- Catch
- Throw
- Throws
- Finally

TRY

- Statements that need to be monitored for exceptions should be placed within **a try** block

CATCH

- If an exception occurs within the try block, it is thrown and Your code can catch this exception using **catch** block and handles it in some rational manner.

THROW

- System generated exception are automatically thrown by the jvm. To manually throw an exception, use the keyword **throw**.

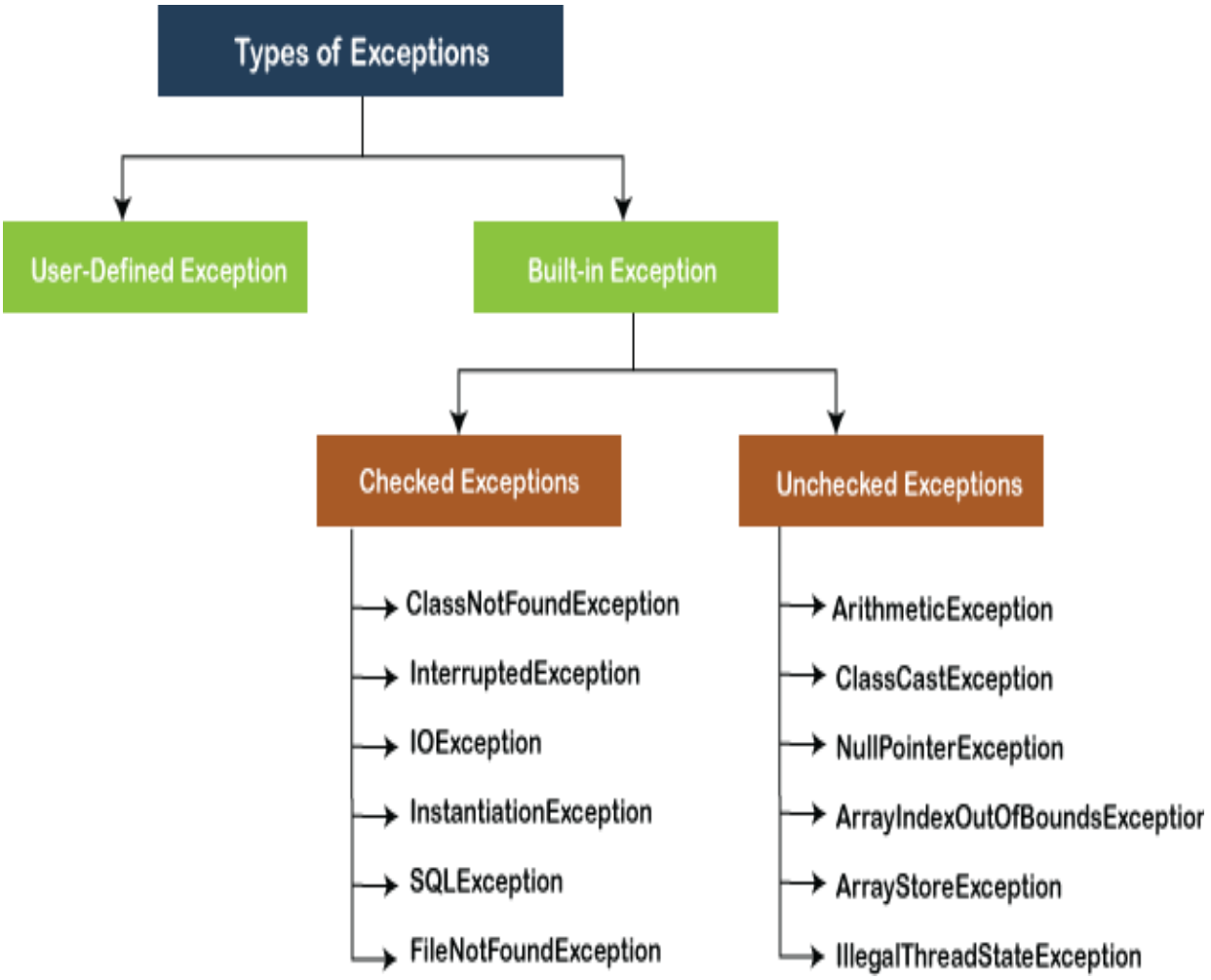
THROWS

- Any exception that is thrown out of method must be specified as such by a **throws** clause

FINALLY

- Any code that obsolutely must be executed after a try block completes isput in a **finally** block.

Hierarchy of Standard Exception Classes



1. User-Defined Exceptions

- These are custom exceptions created by users.
- In Java, users can define their own exceptions by extending the Exception class (for checked exceptions) or RuntimeException (for unchecked exceptions).

2. Built-in Exceptions

Built-in exceptions are predefined in Java and categorized as:

a. Checked Exceptions:

- Checked exceptions must be either caught or declared in the throws clause of a method.
- Examples include:
    - ClassNotFoundException
    - IOException
    - SQLException
    - FileNotFoundException
    -

b. Unchecked Exceptions:

- These exceptions occur at runtime and don't need to be declared in a method's throws clause.
- Examples include:
    - ArithmeticException
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - IllegalArgumentException

Examples of Programs

1. **User-Defined Exception Example**: Here's how you can create and use a user-defined exception.

```
class CustomException extends Exception {
   public CustomException(String message) {
      super(message);
   }
}
public class Main {
   public static void validateAge(int age) throws CustomException {
      if(age < 18) {
         throw new CustomException("Age is less than 18, not eligible.");
      } else {
         System.out.println("Eligible");
      }
   }
   public static void main(String[] args) {
      try {
         validateAge(15);
      } catch (CustomException e) {
         System.out.println("Caught: " + e.getMessage());
      }
   }
}
```

Output:

```
Caught: Age is less than 18, not eligible.
```

b. Unchecked Exceptions:

- These exceptions occur at runtime and don't need to be declared in a method's throws clause.
- Examples include:
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - IllegalArgumentException

Built-in-Exception-Creating own Exceptions

Arithmetic exception

```
class ArithmeticException_Demo {
public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a / b; // cannot divide by zero
            System.out.println("Result = " + c);
        }


        catch (ArithmeticException e) {
```

ArrayIndexOutOfBounds Exception

```
class ArrayIndexOutOfBound_Demo {
public static void main(String args[])
  {
    try {
      int a[] = new int[5];
      a[6] = 9; // accessing 7th element in an array of
      // size 5
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array Index is Out Of Bounds");
    }
  }
```

}

FileNotFoundException

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo {

public static void main(String args[])
    {
        try {

            // Following file does not exist
            File file = new File("E:// file.txt");

            FileReader fr = new FileReader(file);
        }
```

NullPointerException

```
class NullPointer_Demo {
public static void main(String args[])
    {
        try {
            String a = null; // null value
            System.out.println(a.charAt(0));
        }
        catch (NullPointerException e) {
            System.out.println("NullPointerException..");
```

Java File Handling

In Java by reading and writing text and binary files. File handling is crucial for any software developer since it allows you to store and retrieve data, create logs, and process input/output files.

java provides several classes and methods to work with files. The most common classes used for file handling are:

- **File**: Represents a file or directory and provides methods to work with them.
- **FileInputStream** and **FileOutputStream**: Used for reading and writing binary files.
- **FileReader** and **FileWriter**: Used for reading and writing text files.
- **BufferedReader** and **BufferedWriter**: Used for buffered reading and writing.

To read a text file, follow these steps:

1. Create a Fileobject representing the text file.
2. Create a FileReaderobject to read the file.
3. Create a BufferedReaderobject to read text from the file efficiently.
4. Read the file using the readLine()method.
5. Close the BufferedReaderobject.

Types of Streams

Java defines two types of streams:

- **Byte Streams**: Used to perform input and output of 8-bit bytes.
- **Character Streams**: Used to perform input and output for characters (16-bit Unicode).

Byte Streams

Byte streams in Java are used to handle raw binary data. These streams read/write data in the form of bytes. Classes for byte streams are part of the java.io package and typically extend InputStream or OutputStream.

Common Byte Stream Classes:

- **FileInputStream**: Reads bytes from a file.
- **FileOutputStream**: Writes bytes to a file.
- **BufferedInputStream**: Reads bytes from a file with buffering.
- **BufferedOutputStream**: Writes bytes to a file with buffering.

1. FileInputStream

- **Purpose**: Reads raw bytes from a file.
- It is used to read the content of a file byte by byte, making it ideal for reading binary files like images, audio, etc.
- It is part of the java.io package and extends the InputStream class.

Example:

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int data;
            while ((data = fis.read()) != -1) {  // Read byte by byte
                System.out.print((char) data);  // Convert byte to char and print
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. FileOutputStream

- **Purpose**: Writes raw bytes to a file.
- It is used to write data into a file byte by byte, useful for writing binary data.
- It is part of the java.io package and extends the OutputStream class.

Example:

```
import java.io.FileOutputStream;
import java.io.IOException;


public class FileOutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String content = "Hello, World!";
            fos.write(content.getBytes());  // Convert string to bytes and write to file
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The content "Hello, World!" is converted into bytes using the getBytes() method and written to the file output.txt.
- The write() method writes bytes to the file.

3. BufferedInputStream

- **Purpose**: Reads bytes from a file with buffering to improve performance.
- It wraps a FileInputStream and provides buffering, which reduces the number of actual read operations performed on the file, improving efficiency.
- It is part of the java.io package and extends the InputStream class.

Example:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

```
public class BufferedInputStreamExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("example.txt"))) {
            int data;
            while ((data = bis.read()) != -1) {
                System.out.print((char) data);  // Convert byte to char and print
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. BufferedOutputStream

- **Purpose**: Writes bytes to a file with buffering to improve performance.
- It wraps a FileOutputStream and provides buffering, reducing the number of actual write operations performed on the file.
- It is part of the java.io package and extends the OutputStream class.

Example:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class BufferedOutputStreamExample {
    public static void main(String[] args) {
        try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("output.txt"))) {
            String content = "Hello, Buffered World!";
            bos.write(content.getBytes());  // Write bytes to buffer
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Advantages of Buffered Streams:

- **Performance Improvement**: Buffered streams improve performance by reducing the number of disk I/O operations. Instead of reading/writing byte-by-byte, buffered streams work with larger blocks of data.
- **Efficiency**: Buffered streams are more efficient when reading from or writing to slow sources, such as files on a disk or network connections.

Key Differences:

1. **FileInputStream/FileOutputStream**:
    - Read/write data one byte at a time.
    - Suitable for binary data but not optimized for frequent I/O operations.
2. **BufferedInputStream/BufferedOutputStream**:
    - Read/write data in chunks, improving efficiency by reducing I/O operations.
    - Suitable for larger files or when efficiency is a concern.

character Streams

Character Streams handle 16-bit Unicode characters, making them ideal for reading and writing text data. Classes for character streams typically extend the Reader class (for reading) or the Writer class (for writing).

Common Character Stream Classes:

1. **FileReader**: Reads characters from a file.
2. **FileWriter**: Writes characters to a file.
3. **BufferedReader**: Wraps FileReader to provide efficient character buffering while reading text.
4. **BufferedWriter**: Wraps FileWriter to provide efficient character buffering while writing text.

FileReader

- **Purpose**: Reads characters from a file.
- It is a convenient class for reading text files as it reads characters rather than bytes, making it suitable for handling text data.

Example:

```
import java.io.FileReader;

import java.io.IOException;

public class FileReaderExample {

   public static void main(String[] args) {

      try (FileReader fr = new FileReader("example.txt")) {

         int data;
```

```
            while ((data = fr.read()) != -1) {
                System.out.print((char) data);  // Read character-by-character.
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. FileWriter

- **Purpose**: Writes characters to a file.
- FileWriter is used for writing text data to a file, character-by-character. It's a simple way to write text files.

Example:

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            String content = "Hello, FileWriter!";
            fw.write(content);  // Write string to file.
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. BufferedReader

- **Purpose**: Wraps FileReader to provide efficient character buffering while reading text.
- It reads text from a file more efficiently by buffering character input. It also provides convenient methods like readLine() for reading entire lines of text.

Example:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {  // Read line-by-line.
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. BufferedWriter

- **Purpose**: Wraps FileWriter to provide efficient character buffering while writing text.
- It writes text to a file more efficiently by buffering character output. It also provides convenient methods like newLine() to write line separators.

Example:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
            bw.write("Hello, BufferedWriter!");  // Write text to file.
            bw.newLine();  // Insert a new line.
            bw.write("This is the second line.");
            System.out.println("Data written to file successfully.");
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1. Scanner Class

The **Scanner** class in Java is part of the java.util package. It is widely used to parse primitive types (e.g., int, double, float, etc.) and strings using regular expressions. A common use case for the Scanner class is reading input from the user, reading files, or processing input from other data sources like input streams.

Common Uses:

1. **Reading from the Console** (Standard Input)
2. **Reading from Files**

---

a. Reading from the Console (Standard Input)

The Scanner class can read input from the console using the standard input stream (System.in).

Example:

```
import java.util.Scanner;
public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();  // Read a full line of text
        System.out.println("Hello, " + name + "!");

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();  // Read an integer value
        System.out.println("You are " + age + " years old.");
    }
}
```

Explanation:

- The nextLine() method is used to read a full line of text.
- The nextInt() method reads an integer value.

b. Reading from a File

The Scanner class can also be used to read data from a file by passing a File object or the file path to the Scanner constructor.

Example:

```
import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;


public class FileReadingExample {

   public static void main(String[] args) {

      try {

         File file = new File("input.txt");

         Scanner scanner = new Scanner(file);


         while (scanner.hasNextLine()) {

            String line = scanner.nextLine();  // Read line-by-line

            System.out.println(line);

         }


         scanner.close();

      } catch (FileNotFoundException e) {

         e.printStackTrace();

      }

   }

}
```

Explanation:

- The Scanner reads lines from the file input.txt line-by-line using the nextLine() method.
- The hasNextLine() method checks if there are more lines to read.

2. Files Class

The **Files** class in Java is part of the java.nio.file package, which provides a variety of utility methods for file handling, including reading, writing, creating, copying, moving, and deleting files and directories. It supports working with **Path** objects, which represent file and directory locations in the file system.

Common Operations:

1. **Reading a File**
2. **Writing to a File**
3. **Copying Files**
4. **Deleting Files**
5. **Creating Files and Directories**