EXCEPTION HANDLING AND COLLECTION FRAMEWORK

1. Write a Java program to implement simple exception handling
2. Write a Java program to implement arithmetic exception
3. Write a Java program to use finally block in exception handling
4. Write a Java program to use multiple catch block
5. Write a Java program to use throw keyword
6. Write a java program to use throws keyword
7. Write a Java program to implement custom exception
8. Write a Java program to implement exception propagation
9. Write a Java program to implement exception chaining
10. Write a Java program to use simple inner class in your program
11. Write a Java program to use static inner class
12. Write a Java program to use local inner class
13. Write a Java program to use nested interface
14. Write a Java program to display date in different format
15. Write a Java program to display different calendar information using calendar class
16. Write a Java program to add substract a days/month into current date and time
17. Write a Java program to use a Gregorian calendar to display calendar information

Write a Java program to implement simple exception handling

Code:

```
        public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int numerator = 10;
            int denominator = 0;
            int result = numerator / denominator; // This will throw an ArithmeticException
            System.out.println("Result: " + result); // This line won't be executed
        } catch (ArithmeticException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Output:

```
        An error occurred: / by zero
```

Write a Java program to implement arithmetic exception

Code:

```
        try {
   int numerator = 10;
   int denominator = 0;
   int result = numerator / denominator; // This will throw an ArithmeticException
   System.out.println("Result: " + result); // This line won't be executed
} catch (ArithmeticException e) {
   System.out.println("An error occurred: " + e.getMessage());
}
```

Here, `e` represents an instance of the `ArithmeticException` class. You can use methods of the `e` object to access information about the exception, such as the error message using `e.getMessage()`. This allows you to handle the exception and provide useful information to the user or log it for debugging purposes.

Write a Java program to use finally block in exception handling

Code:

```
public class FinallyBlockExample {
   public static void main(String[] args) {
      try {
         int numerator = 10;
         int denominator = 0;
         int result = numerator / denominator; // This will throw an
ArithmeticException
         System.out.println("Result: " + result); // This line won't be executed
      } catch (ArithmeticException e) {
         System.out.println("An error occurred: " + e.getMessage());
      } finally {
         System.out.println("Finally block is executed regardless of exception");
      }
   }
}
```

Output:

```
An error occurred: / by zero
Finally block is executed regardless of exception
```

Write a Java program to use multiple catch block

Code:

```
public class MultipleCatchBlockExample {

public static void main(String[] args) {
    try {
        int[] numbers = { 1, 2, 3 };
        System.out.println("Number at index 3: " + numbers[3]); // This will
throw an ArrayIndexOutOfBoundsException
        int result = 10 / 0; // This will throw an ArithmeticException
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out of bounds: " + e.getMessage());
    } catch (ArithmeticException e) {
        System.out.println("Arithmetic error: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("Some other error occurred: " + e.getMessage());
    }
}
}
```

Output :

```
Array index out of bounds: Index 3 out of bounds for length 3
```

Write a Java program to use through keyword

Code:

```
public class ThrowKeywordExample {
    public static void main(String[] args) {
        try {
            int age = 15;
            if (age < 18) {
                throw new IllegalArgumentException("Age must be 18 or older");
            }
            System.out.println("You are eligible to vote!");
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

In this program, we're checking the value of `age` and throwing an `IllegalArgumentException` if the age is less than 18. The `throw` keyword is used to explicitly throw an exception. The `catch` block then handles the thrown exception by printing an error message.

Output:

```
Error: Age must be 18 or older
```

Here's a Java program that demonstrates the use of the `throws` keyword to declare exceptions that a method might throw:

```
public class ThrowsKeywordExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // This will throw an ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }

    public static int divide(int numerator, int denominator) throws
ArithmeticException {
        if (denominator == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return numerator / denominator;
    }
}
```

In this program, the `divide` method declares that it might throw an `ArithmeticException` using the `throws` keyword. This informs the caller that they should handle or propagate the exception. When calling the `divide` method in the `main` method, the caller can choose to catch the exception or let it propagate further. In this example, the `main` method catches the exception and prints an error message.

In Java, both the `throw` and `throws` keywords are related to exception handling, but they serve different purposes:

1**. `throw` keyword:**

- The `throw` keyword is used to explicitly throw an exception within a method.

- It is used to create and throw instances of exception classes when a specific condition is met.

- When you use `throw`, you are generating an exception manually and causing the program to handle it accordingly.

- It is used within the method body to indicate that a specific exceptional situation has occurred.

2. **`throws` keyword**:

- The `throws` keyword is used in the method declaration to indicate that the method might throw one or more specific exceptions.

- It is used to delegate the responsibility of handling exceptions to the caller of the method.

- When you use `throws`, you are declaring that your method could potentially throw the specified exceptions, and you are transferring the responsibility of handling them to the calling code.

- It is used in the method signature followed by the names of the exception classes that the method might throw.

Here's a simple example to illustrate the difference:

```
java
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            divideByZero(); // This method throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught exception: " + e.getMessage());
```

```
        }
    }

    public static void divideByZero() {
        throw new ArithmeticException("Division by zero");
            }
}
```

In this example, the `divideByZero` method uses the `throw` keyword to manually throw an `ArithmeticException`. The method in the `main` function calls `divideByZero`, and since it knows that the method throws an exception, it catches and handles the exception.

Remember:

- `throw` is used within a method to manually throw an exception.

- `throws` is used in the method declaration to indicate that the method might throw specific exceptions, transferring the responsibility of handling them to the caller.

Write a java program to use throws keyword

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

Write a Java program to implement custom exception

Code:

```
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            int age = 15;
            if (age < 18) {
                throw new CustomException("Age must be 18 or older");
            }
            System.out.println("You are eligible to vote!");
        } catch (CustomException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Error: Age must be 18 or older
```

Or

You are eligible to vote! (if your age is above 18)

Write a Java program to implement exception propagation

Code:

```
public class ExceptionPropagationExample {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("Exception caught in main: " + e.getMessage());
        }
    }

    static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            System.out.println("Exception caught in method1: " + e.getMessage());
            throw e; // Propagate the exception up the call stack
        }
    }

    static void method2() throws Exception {
        try {
            int result = 10 / 0; // This will throw an ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught in method2: " + e.getMessage());
```

```
            throw new Exception("Exception in method2", e); // Propagate a new exception up the call stack
        }
    }
}
```

Output:

```
            Exception caught in method2: / by zero
Exception caught in method1: Exception in method2
Exception caught in main: Exception in method2
```

Write a Java program to implement exception chaining

Code:

```
class CustomException extends Exception {
  public CustomException(String message, Throwable cause) {
    super(message, cause);
  }
}

public class ExceptionChainingExample {
  public static void main(String[] args) {
    try {
      method1();
    } catch (CustomException e) {
      System.out.println("Exception caught in main: " + e.getMessage());
      System.out.println("Original cause: " + e.getCause().getMessage());
    }
  }

  static void method1() throws CustomException {
    try {
      method2();
    } catch (Exception e) {
      throw new CustomException("Custom exception in method1", e);
    }
  }

  static void method2() throws Exception {
```

```
        try {
            int result = 10 / 0; // This will throw an ArithmeticException
        } catch (ArithmeticException e) {
            throw new Exception("Exception in method2", e);
        }
    }
}
```

In this example, we have a custom exception class `CustomException` that extends the base `Exception` class. This custom exception class has a constructor that takes a message and a cause (`Throwable`) as arguments and passes them to the superclass constructor using `super(message, cause)`.

The program demonstrates exception chaining by catching an exception in `method2`, then wrapping it with a new exception (`CustomException`) in `method1`, which is again caught and handled in the `main` method. The original cause of the exception is also retained and can be accessed using the `getCause()` method.

This chaining mechanism helps to provide more context and information about the origin of the exception, which can be useful for debugging and understanding the flow of exceptions.

In Java, the term "propagate" is used to describe the process of passing an exception from one method or block of code to another, usually up the call stack. When an exception occurs in a method and is not caught and handled within that method, it can be propagated to higher levels of the program to find a suitable catch block or be reported to the user.

Here's how exception propagation works:

1. An exception is thrown within a method due to some exceptional condition, such as a division by zero or array index out of bounds.

2. If the exception is not caught and handled within the method using a `catch` block, it is propagated to the calling method.

3. This process continues up the call stack until an appropriate `catch` block is found that can handle the exception or until the exception reaches the top level of the program, resulting in the program terminating and displaying an error message.

Propagation allows you to separate the responsibility of handling exceptions. Instead of every method needing to know how to handle every possible exception, you can have specific catch blocks or exception-handling strategies at higher levels of the program.

Here's a simple example of exception propagation:

```
java
public class PropagationExample {
   public static void main(String[] args) {
      try {
         method1();
      } catch (Exception e) {
         System.out.println("Exception  caught in main: " + e.getMessage());
      }
   }
```

```
    static void method1() throws Exception {
        method2(); // Exception propagates to method1
    }

    static void method2() throws Exception {
        int result = 10 / 0; // This will throw an ArithmeticException
    }
}
```

In this example, the exception is thrown in `method2` and propagates up to `method1`, and finally, it's caught and handled in the `main` method. This demonstrates the process of exception propagation in Java.

Write a Java program to use simple inner class in your program

Code:

```
        public class OuterClass {
    private int outerValue = 10;

    // Inner class definition
    class InnerClass {
        void display() {
            System.out.println("Inner  class value: " + outerValue);
        }
    }

    public static void main1(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

In this program, `InnerClass` is an inner class within the `OuterClass`. The `InnerClass` can access the private member `outerValue` of the `OuterClass`. The `main` method demonstrates how to create an instance of the `InnerClass` and call its `display` method.

Output:

```
        Inner class value: 10
```

Write a Java program to use static inner class

Code:

```
public class OuterClass {
    private static int outerValue = 10;

    // Static inner class definition
    static class StaticInnerClass {
        void display() {
            System.out.println("Static  inner class value: " + outerValue);
        }
    }

    public static void main(String[] args) {
        OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();
        inner.display();
    }
}
```

Output:

```
Static inner class value: 10
```

A static inner class in Java can access both static and non-static members (variables and methods) of its outer class. However, it's important to understand the rules and limitations:

1. ***Static Members:*** A static inner class can directly access static variables and methods of its outer class. This includes static fields and methods, regardless of their access modifiers (public, protected, package-private, or private).

2. ***Non-Static Members:*** A static inner class cannot directly access non-static (instance) variables and methods of its outer class. This is because a static inner class exists independently of any instances of the outer class. To access non-static members, you would need to create an instance of the outer class.

Here's a summary:

- Accessible from a static inner class:
  - Static fields
  - Static methods

- Not directly accessible from a static inner class:
  - Instance (non-static) fields
  - Instance (non-static) methods

If you need to access non-static members from a static inner class, you can do so by creating an instance of the outer class within the static inner class.

Remember that these rules are related to the scope and accessibility of members within the context of an inner class.

To create an instance of the outer class within a static inner class, you can follow these steps:

1. Import the outer class if it's in a different package.
2. Use the syntax `OuterClassName.this` to reference the instance of the outer class.

Here's an example:

```
java
public class OuterClass {
    private int outerValue = 10;

    // Static inner class definition
    static class StaticInnerClass {
        void displayOuterValue() {
            OuterClass outerInstance = OuterClass.this;
            System.out.println("Outer class value from inner class: " +
outerInstance.outerValue);
        }
    }

    public static void main(String[] args) {
        OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();
        inner.displayOuterValue();
    }
}
```

In this example, the `StaticInnerClass` uses `OuterClass.this` to refer to an instance of the outer class. This allows you to access the non-static member `outerValue` from within the static inner class. Note that this approach only works if the inner class is a static inner class. If it were a non-static inner class (also known as an inner class), you wouldn't need this syntax to access the outer class instance.

Write a Java program to use local inner class

Code:

```
public class OuterClass {
  private int outerValue = 10;

  public void outerMethod() {
    // Local inner class definition within a method
    class LocalInnerClass {
      void display() {
        System.out.println("Local inner class value: " + outerValue);
      }
    }

    // Creating an instance of the local inner class and calling its method
    LocalInnerClass inner = new LocalInnerClass();
    inner.display();
  }
  public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    outer.outerMethod();
  }
}
```

In this program, the `LocalInnerClass` is defined within the `outerMethod()` of the `OuterClass`. This makes the `LocalInnerClass` a local inner class. Local inner classes can only be used within the method where they are defined. The `outerMethod()` creates an instance of the `LocalInnerClass` and calls its `display()` method to show the usage of the local inner class.

Write a Java program to use nested interface

Code:

```
public class NestedInterfaceExample {
    public interface OuterInterface {
        void outerMethod();

        interface InnerInterface {
            void innerMethod();
        }
    }
    public static class MyClass implements OuterInterface.InnerInterface {
        @Override
        public void innerMethod() {
            System.out.println("Inner interface method implementation");
        }
    }
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        myClass.innerMethod();
    }
}
```

In this program, we have a nested interface structure:

- `OuterInterface` is an outer interface.
- `InnerInterface` is a nested interface within `OuterInterface`.

Then, we have the `MyClass` class implementing the `OuterInterface.InnerInterface` interface. The `main` method creates an instance of `MyClass`.

Write a Java program to display date in different format

Code:

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateFormatExample {
    public static void main(String[] args) {
        Date currentDate = new Date();

        // Define different date formats
        SimpleDateFormat dateFormat1 = new SimpleDateFormat("dd-MM-yyyy");
        SimpleDateFormat dateFormat2 = new SimpleDateFormat("MMMM dd, yyyy");
        SimpleDateFormat dateFormat3 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        // Format and display the date using different formats
        String formattedDate1 = dateFormat1.format(currentDate);
        String formattedDate2 = dateFormat2.format(currentDate);
        String formattedDate3 = dateFormat3.format(currentDate);

        System.out.println("Date in dd-MM-yyyy format: " + formattedDate1);
        System.out.println("Date in MMMM dd, yyyy format: " + formattedDate2);
        System.out.println("Date in yyyy-MM-dd HH:mm:ss format: " + formattedDate3);
    }
}
```

Output:

```
Date in dd-MM-yyyy format: 28-08-2023
Date in MMMM dd, yyyy format: August 28, 2023
Date in yyyy-MM-dd HH:mm:ss format: 2023-08-28 19:54:37
```

Write a Java program to display different calendar information using calendar class

Code:

```
import java.util.Calendar;


public class CalendarExample {
   public static void main(String[] args) {
      // Get a Calendar instance
      Calendar calendar = Calendar.getInstance();


      // Get current date and time
      int year = calendar.get(Calendar.YEAR);
      int month = calendar.get(Calendar.MONTH) + 1; // Months are 0-based, so add 1
      int day = calendar.get(Calendar.DAY_OF_MONTH);
      int hour = calendar.get(Calendar.HOUR_OF_DAY);
      int minute = calendar.get(Calendar.MINUTE);
      int second = calendar.get(Calendar.SECOND);


      // Display the information
      System.out.println("Year: " + year);
      System.out.println("Month: " + month);
      System.out.println("Day: " + day);
      System.out.println("Hour: " + hour);
      System.out.println("Minute: " + minute);
      System.out.println("Second: " + second);
   }
}
```

Output:

Year: 2023

Month: 8

Day: 28

Hour: 20

Minute: 1

Second: 21

The `getInstance()` method is a static factory method provided by the `Calendar` class in Java. It's used to create an instance of the `Calendar` class representing the current date and time according to the default time zone and locale.

Here's how the method works:

java

```
Calendar calendar = Calendar.getInstance();
```

When you call `Calendar.getInstance()`, it returns an instance of the `Calendar` class with the current date and time settings. This method is often used as a way to obtain a `Calendar` object to work with date and time calculations, formatting, or parsing.

It's worth noting that since Java 8, a more modern and recommended approach for working with dates and times is to use the `java.time` package (also known as the Java Date and Time API), which offers improved functionality and better support for modern date and time operations.

Write a Java program to add substract a days/month into current date and time

Code:

```
import java.util.Calendar;
import java.util.Date;

public class DateManipulationExample {
   public static void main(String[] args) {
      // Get the current date and time
      Calendar calendar = Calendar.getInstance();

      // Adding and subtracting days
      int daysToAdd = 10;
      int daysToSubtract = 5;

      calendar.add(Calendar.DAY_OF_MONTH, daysToAdd);
      Date futureDate = calendar.getTime();

      calendar.add(Calendar.DAY_OF_MONTH, -daysToSubtract);
      Date pastDate = calendar.getTime();

      // Adding and subtracting months
      int monthsToAdd = 2;
      int monthsToSubtract = 1;

      calendar.setTime(futureDate);
      calendar.add(Calendar.MONTH, monthsToAdd);
```

```
        Date futureMonth  = calendar.getTime();

        calendar.setTime(pastDate);
        calendar.add(Calendar.MONTH, -monthsToSubtract);
        Date pastMonth  = calendar.getTime();

        // Display the manipulated dates
        System.out.println("Future  date after adding days: " + futureDate);
        System.out.println("Past  date after subtracting days: " + pastDate);
        System.out.println("Future  date after adding months: " + futureMonth);
        System.out.println("Past  date after subtracting months:  " + pastMonth);
    }
}
```

In this program, we first obtain the current date and time using `Calendar.getInstance()`. Then, we demonstrate adding and subtracting days by using the `add(Calendar.DAY_OF_MONTH, amount)` method. We also demonstrate adding and subtracting months by using the `add(Calendar.MONTH, amount)` method.

Remember that the `Calendar` class performs these operations in place, modifying the existing `Calendar` object. Finally, we use the `getTime()` method to convert the `Calendar` object to a `Date` object for display.

Write a Java program to use a Gregorian calendar to display calendar information

Code:

```
import java.util.GregorianCalendar;
import java.util.Calendar;

public class GregorianCalendarExample {
    public static void main(String[] args) {
        // Create a GregorianCalendar instance
        GregorianCalendar calendar = new GregorianCalendar();

        // Get current date and time
        int year = calendar.get(Calendar.YEAR);
        int month = calendar.get(Calendar.MONTH) + 1; // Months are 0-based,
so add 1
        int day = calendar.get(Calendar.DAY_OF_MONTH);
        int hour = calendar.get(Calendar.HOUR_OF_DAY);
        int minute = calendar.get(Calendar.MINUTE);
        int second = calendar.get(Calendar.SECOND);

        // Display the information
        System.out.println("Year: " + year);
        System.out.println("Month: " + month);
        System.out.println("Day: " + day);
        System.out.println("Hour: " + hour);
        System.out.println("Minute: " + minute);
        System.out.println("Second: " + second);
    }
}
```

Output:

Year: 2023

Month: 8

Day: 28

Hour: 20

Minute: 1

Second: 21

The `GregorianCalendar` is a concrete implementation of the `java.util.Calendar` abstract class in Java. It's used to work with dates and times based on the Gregorian calendar system, which is the calendar system most commonly used today.

The Gregorian calendar system was introduced by Pope Gregory XIII in 1582 as a reform of the Julian calendar. It's the calendar system that is widely used in most parts of the world for civil purposes.

The `GregorianCalendar` class provides methods for performing various date and time calculations, formatting and parsing dates, and working with time zones. It allows you to manipulate dates and times by adding or subtracting years, months, days, hours, minutes, and seconds.

Here are some key points about the `GregorianCalendar` class:

- It provides a way to represent and manipulate dates and times based on the Gregorian calendar system.
- It's a subclass of the `Calendar` class, which provides a common interface for various calendar systems.
- It allows you to work with dates and times, perform calculations, and convert between various units.
- It supports handling time zone differences and daylight saving time adjustments.