1.Interface definition,syntax, example program to implement interface

Definition:

- An interface in Java is a blueprint of a class.
- It contains only abstract methods, constant declarations, default methods, and static methods (from Java 8 onwards).
- Interfaces allow for achieving abstraction, defining a contract that classes must follow when they implement the interface.

Syntax:

```
interface <interface_name>{
  // declare constant fields
  // declare methods that abstract
   // by default.
}
```

Program:

```
interface MyInterface {
    void myMethod();
}

class MyClass implements MyInterface {
    public void myMethod() {
        System.out.println("Implemented myMethod in MyClass");
    }
}

// Main method to demonstrate interface implementation
public class InterfaceExample {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.myMethod();
    }
}
```

2.abstract class definition,syntax, example program to implement abstract class.

Definition:

"Abstract" refers to something that is conceptual or theoretical rather than concrete or specific. In programming, "abstract" often describes something that lacks implementation details or is meant to be further defined by subclasses.

Syntax:

[access modifier] abstract class ClassName {

// Variables, constructors, methods

// Abstract method declaration

abstract void methodName();

}

Program:

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

3.In exception handling

*Try,catch, throw,throws, finally keywords usage, syntax
*User defined exception
*Nested try catch
*Types of exceptions-checked, unchecked,chained, uncaught, errors

Keywords and Syntax:

- try : Used to enclose the code where exceptions might occur.
- catch : Catches and handles the exceptions that occur in the `try` block.
- throw : Explicitly throws an exception within a method or code block.
- throws : Used in method signature to declare exceptions that might be thrown by that method.
- finally : Contains code that always executes, whether an exception is thrown or not.

Syntax:

```
try {
    // Code where exception might occur
} catch (ExceptionType1 e1) {
    // Handling for ExceptionType1
} catch (ExceptionType2 e2) {
    // Handling for ExceptionType2
} finally {
    // Cleanup or closing resources
}
```

User Defined Exception:

Definition:
A user-defined exception is created by extending the `Exception` class or its subclasses to define custom exceptions catering to specific conditions in a program.

Example:
```java
class CustomException extends Exception {
    // Constructor for custom exception
    CustomException(String message) {
        super(message);
    }
}

class MyProgram {
    // Usage of custom exception
    void someMethod() throws CustomException {
        if(someCondition) {
            throw new CustomException("Custom Exception Message");
        }
    }
}
```

#Nested `try-catch` Blocks:

Definition:
Nested `try-catch` blocks are used to handle exceptions in a hierarchical manner, allowing for more specific handling of exceptions.

Syntax:
```java
try {
    // Outer try block
    try {
        // Inner try block
    } catch (ExceptionType1 e1) {
        // Inner catch block for ExceptionType1
    }
} catch (ExceptionType2 e2) {
    // Outer catch block for ExceptionType2
}
```

Types of Exceptions:

Checked Exceptions:
- Checked at compile-time.
- Must be either caught using `try-catch` blocks or declared using `throws` in the method signature.

Unchecked Exceptions:
- Not checked at compile-time (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`).
- Don't need to be explicitly handled or declared.

Chained Exceptions:
- Refers to an exception that is caused by another exception.
- `getCause()` method can retrieve the root cause of the exception.

Uncaught Exceptions:
- Exceptions that are not caught or handled within the code.
- Can lead to program termination or abnormal behavior.

Errors:
- Irrecoverable issues such as `OutOfMemoryError`, `StackOverflowError`.
- Not meant to be caught or handled as they typically indicate serious problems beyond the application's control.

4.In polymorphism

Comparison between compile time and run time polymorphism

Syntax and example program for both.

Compile-Time Polymorphism (Method Overloading):

Definition:

Compile-time polymorphism or method overloading occurs when there are multiple methods with the same name within the same class but with different parameters.

Compile-Time Polymorphism:

Determined during compile-time based on method signatures.

Syntax:

```
class MyClass {
   void myMethod(int num) {
      // Method implementation
   }

   void myMethod(int num1, int num2) {
      // Method implementation with different parameters
   }
}
```

Example Program (Method Overloading):

```
class MathOperations {
   int add(int a, int b) {
      return a + b;
   }

   double add(double a, double b) {
      return a + b;
   }
}
```

In this example, `MathOperations` class has two `add()` methods with the same name but different parameter types (integers and doubles). The appropriate method is selected at compile-time based on the method signature and the arguments passed.

Run-Time Polymorphism (Method Overriding):

Definition:

Run-time polymorphism or method overriding occurs when a subclass provides a specific implementation of a method that is already present in its superclass, with the same method signature.

Run-Time Polymorphism:

Determined during runtime based on the actual object being referred to (method of the actual object is invoked).

Syntax:

```
class Superclass {
   void myMethod() {
      // Method implementation
   }
}

class Subclass extends Superclass {
   @Override
   void myMethod() {
      // Subclass-specific method implementation
   }
}
```

Example Program (Method Overriding):

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   @Override
   void sound() {
      System.out.println("Dog barks");
   }
}
```

In this example, `Animal` class has a `sound()` method, and `Dog` class extends `Animal` and overrides the `sound()` method with a specific implementation. The method called on an object of `Dog` will execute the overridden method at runtime.

5.In multithreading
* Thread life cycle
*Ways of implementing threads,one example program for each.
*Thread priority predefined methods, example program
*Synchronization definition, syntax, example program

Thread Lifecycle:

1. **New:** When a thread is created but not yet started.
2. **Runnable:** When a thread is ready to run and waiting for CPU time.
3. **Running:** When the thread gets CPU time and is executing.
4. **Blocked/Waiting:** When a thread is temporarily inactive.
5. **Terminated/Dead:** When the thread finishes its execution or is stopped.

Ways of Implementing Threads:

Extending the `Thread` Class:

```
class MyThread extends Thread {
    public void run() {
        // Thread execution logic
    }
}

// Creating and starting the thread
MyThread thread = new MyThread();
thread.start();
```

Implementing `Runnable` Interface:

```
class MyRunnable implements Runnable {
    public void run() {
        // Thread execution logic
    }
}

// Creating a thread using Runnable
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Thread Priority and Predefined Methods:

Thread Priority Methods:
- `getPriority()`: Retrieves the priority of a thread.
- `setPriority(int priority)`: Sets the priority of a thread.

Example Program for Thread Priority:

```
class PriorityThread extends Thread {
    public void run() {
        System.out.println("Thread Priority: " + Thread.currentThread().getPriority());
    }
}

public class PriorityExample {
    public static void main(String[] args) {
        PriorityThread thread1 = new PriorityThread();
        PriorityThread thread2 = new PriorityThread();

        thread1.setPriority(Thread.MIN_PRIORITY); // Setting minimum priority
        thread2.setPriority(Thread.MAX_PRIORITY); // Setting maximum priority

        thread1.start();
        thread2.start();
    }
}
```

Synchronization:

Definition:
Synchronization in Java ensures that only one thread can access a shared resource at a time to prevent data inconsistency.

Syntax:

```
class SharedResource {
    synchronized void sharedMethod() {
        // Synchronized method block
    }

    void nonSharedMethod() {
        synchronized (this) {
            // Synchronized block using 'this'
        }
    }
}
```

Example Program for Synchronization:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}

public class SynchronizationExample {
    public static void main(String[] args) {
        Counter counter = new Counter();

        // Multiple threads accessing shared resource
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
```

```
        }
    });

    Thread thread2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    });

    thread1.start();
    thread2.start();

    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Count: " + counter.getCount());
  }
}
```