Exception Handling and Multithreading

Concepts of Exception Handling

1. **Exception Handling in Java**
   - **Definition**: Mechanism to handle runtime errors, ensuring normal flow of application execution.
   - **Hierarchy**:
     - `Throwable`
       - `Error` : Serious problems that applications should not try to catch.
       - `Exception` : Conditions that applications might want to catch.
         - `RuntimeException` : Unchecked exceptions, typically caused by programming errors.
         - `Checked Exceptions` : Must be either caught or declared in the method signature.
   - **Keywords**:
     - `try` : Block of code to monitor for exceptions.
     - `catch` : Block of code that handles exceptions.
     - `finally` : Block of code that executes after try-catch, regardless of whether an exception was thrown.
     - `throw` : Used to explicitly throw an exception.
     - `throws` : Used in method signature to declare exceptions that can be thrown.
2. **Syntax and Usage**:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle exception of type ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle exception of type ExceptionType2
} finally {
    // Code to be executed after try or catch blocks
}
```

3. **Custom Exceptions**:
   - Creating a custom exception by extending the `Exception` class:

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

4. **Common Exceptions**:
    - `NullPointerException`
    - `ArrayIndexOutOfBoundsException`
    - `ClassNotFoundException`
    - `IOException`

Benefits of Exception Handling

1. **Separates Error-Handling Code from Regular Code**:
    - Makes the code cleaner and easier to understand.
    - Error-handling code is centralized, reducing redundancy.
2. **Propagates Errors Up the Call Stack**:
    - Allows errors to be handled at a higher level in the application.
    - Can specify at which level the error should be handled.
3. **Group and Differentiate Error Types**:
    - Multiple catch blocks allow different types of exceptions to be handled differently.
    - Specific handlers can be implemented for different types of errors.
4. **Ensures Program Continuity**:
    - Helps maintain normal application flow by handling unexpected events.
    - `finally` block ensures resource release (e.g., closing files, releasing locks) regardless of exception occurrence.
5. **Improves Fault Tolerance**:
    - Enhances the robustness and reliability of the application.
    - Facilitates debugging and maintenance by providing clear error reporting.
6. **Better Resource Management**:
    - Ensures resources like files, network connections, etc., are properly closed or released.
    - `try-with-resources` statement in Java 7 ensures resources are closed automatically.

```
try (ResourceType resource = new ResourceType()) {
    // Use the resource
} catch (ExceptionType e) {
    // Handle the exception
}
```

Exception Hierarchy and Usage of Keywords

1. **Exception Hierarchy**:
    - **Throwable**: The superclass of all errors and exceptions in Java.
        - **Error**: Indicates serious problems that a reasonable application should not try to catch.
            - Example: `OutOfMemoryError` , `StackOverflowError`
        - **Exception**: Indicates conditions that a reasonable application might want to catch.
            - **RuntimeException**: Unchecked exceptions, typically due to programming errors.
                - Example: `NullPointerException` , `ArrayIndexOutOfBoundsException`
            - **Checked Exceptions**: Must be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.
                - Example: `IOException` , `SQLException`

```
java.lang.Object
    ↳ java.lang.Throwable
        ↳ java.lang.Error
        ↳ java.lang.Exception
            ↳ java.lang.RuntimeException
```

2. **Usage of Keywords**:
    - **try**: Block of code where exceptions can occur.
    - **catch**: Block of code to handle the exceptions thrown by the try block.
    - **finally**: Block of code that executes after try/catch blocks, regardless of whether an exception was thrown or caught.
    - **throw**: Used to explicitly throw an exception.
    - **throws**: Used in method signatures to declare exceptions that can be thrown by the method.

```
public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " +
e.getMessage());
        } finally {
            System.out.println("Execution completed.");
        }
    }

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

Built-in Exceptions and Creating Custom Exceptions

1. **Built-in Exceptions**:
    - **ArithmeticException**: Thrown when an exceptional arithmetic condition has occurred.
    - **ArrayIndexOutOfBoundsException**: Thrown to indicate that an array has been accessed with an illegal index.
    - **ClassNotFoundException**: Thrown when an application tries to load a class through its string name but no definition for the class with the specified name could be found.
    - **FileNotFoundException**: Thrown when an attempt to open the file denoted by a specified pathname has failed.
    - **IOException**: Signals that an I/O exception of some sort has occurred.
    - **NullPointerException**: Thrown when an application attempts to use `null` in a case where an object is required.
    - **NumberFormatException**: Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

```
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index is out of bounds: " +
e.getMessage());
}
```

2. **Creating Custom Exceptions**:

- To create a custom exception, extend the `Exception` class (for checked exceptions) or the `RuntimeException` class (for unchecked exceptions).

```
// Custom checked exception
public class CustomCheckedException extends Exception {
    public CustomCheckedException(String message) {
        super(message);
    }
}

// Custom unchecked exception
public class CustomUncheckedException extends RuntimeException {
    public CustomUncheckedException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (CustomCheckedException e) {
            System.out.println("Caught custom checked exception: " +
e.getMessage());
        }

        try {
            validateName(null);
        } catch (CustomUncheckedException e) {
            System.out.println("Caught custom unchecked exception: " +
e.getMessage());
        }
    }

    public static void validateAge(int age) throws
CustomCheckedException {
        if (age < 18) {
            throw new CustomCheckedException("Age must be at least
18");
        }
    }

    public static void validateName(String name) {
        if (name == null) {
            throw new CustomUncheckedException("Name cannot be null");
        }
    }
}
```

String Handling and Exploring `java.util`

1. **String Handling in Java**:
    - **String Class**: Immutable sequence of characters.
        - Common Methods:
            - `length()` : Returns the length of the string.
            - `charAt(int index)` : Returns the character at the specified index.
            - `substring(int beginIndex, int endIndex)` : Returns a new string that is a substring of the original.
            - `concat(String str)` : Concatenates the specified string to the end of the original string.
            - `equals(Object obj)` : Compares this string to the specified object.
            - `compareTo(String anotherString)` : Compares two strings lexicographically.
            - `toUpperCase()` , `toLowerCase()` : Converts all characters in the string to uppercase or lowercase.
            - `trim()` : Removes whitespace from both ends of the string.
            - `replace(char oldChar, char newChar)` : Returns a new string resulting from replacing all occurrences of oldChar in the string with newChar.

    ```java
    String str = "Hello, World!";
    System.out.println("Length: " + str.length());
    System.out.println("Character at index 1: " + str.charAt(1));
    System.out.println("Substring (0, 5): " + str.substring(0, 5));
    System.out.println("Concatenated String: " + str.concat(" Welcome!"));
    System.out.println("Equals 'Hello, World!': " + str.equals("Hello, World!"));
    System.out.println("Compare to 'Hello, Java!': " + str.compareTo("Hello, Java!"));
    System.out.println("Uppercase: " + str.toUpperCase());
    System.out.println("Lowercase: " + str.toLowerCase());
    System.out.println("Trimmed String: " + str.trim());
    System.out.println("Replaced 'o' with 'a': " + str.replace('o', 'a'));
    ```

    - **StringBuilder and StringBuffer**: Mutable sequences of characters.

- `StringBuilder` is not synchronized (faster, not thread-safe).
- `StringBuffer` is synchronized (slower, thread-safe).

```java
StringBuilder sb = new StringBuilder("Hello");
sb.append(", World!");
System.out.println(sb.toString()); // Output: Hello, World!

StringBuffer sbf = new StringBuffer("Hello");
sbf.append(", World!");
System.out.println(sbf.toString()); // Output: Hello, World!
```

2. **Exploring** `java.util`:
   - **Common Classes and Interfaces**:
     - **ArrayList**: Resizable array implementation of the List interface.
     - **HashMap**: Hash table-based implementation of the Map interface.
     - **HashSet**: Hash table-based implementation of the Set interface.
     - **LinkedList**: Doubly-linked list implementation of the List and Deque interfaces.
     - **Stack**: Last-in, first-out (LIFO) stack of objects.
     - **Queue**: Collection designed for holding elements prior to processing.

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Stack;
import java.util.Queue;
import java.util.PriorityQueue;

// ArrayList example
ArrayList<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
System.out.println("ArrayList: " + arrayList);

// HashMap example
HashMap<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "One");
hashMap.put(2, "Two");
System.out.println("HashMap: " + hashMap);

// HashSet example
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Red");
hashSet.add("Green");
System.out.println("HashSet: " + hashSet);
```

```
    // LinkedList example
    LinkedList<String> linkedList = new LinkedList<>();
    linkedList.add("First");
    linkedList.add("Second");
    System.out.println("LinkedList: " + linkedList);

    // Stack example
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    System.out.println("Stack: " + stack);
    System.out.println("Popped element: " + stack.pop());

    // Queue example
    Queue<String> queue = new PriorityQueue<>();
    queue.add("First");
    queue.add("Second");
    System.out.println("Queue: " + queue);
    System.out.println("Polled element: " + queue.poll());
```

Differences Between Multithreading and Multitasking

1. **Multithreading**:
    - **Definition**: The ability of a CPU or a single core in a multi-core processor to execute multiple threads concurrently.
    - **Purpose**: To perform multiple tasks within a single process simultaneously.
    - **Example**: A web server handling multiple requests from different clients concurrently.
    - **Granularity**: Finer, as it involves splitting a single process into multiple threads.
    - **Resource Sharing**: Threads within the same process share the same memory space and resources.
    - **Overhead**: Lower overhead compared to multitasking since threads share the same process resources.

```
public class MultithreadingDemo implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is
running");
    }

    public static void main(String[] args) {
        Thread thread1 = new Thread(new MultithreadingDemo());
        Thread thread2 = new Thread(new MultithreadingDemo());
        thread1.start();
        thread2.start();
    }
}
```

2. **Multitasking**:
    - **Definition**: The ability of an operating system to execute multiple tasks (processes) simultaneously.
    - **Purpose**: To improve the efficiency and performance of the system by allowing multiple processes to run concurrently.
    - **Example**: Running a web browser, a text editor, and a media player simultaneously.
    - **Granularity**: Coarser, as it involves managing multiple independent processes.
    - **Resource Sharing**: Each process has its own memory space and resources; communication between processes typically requires inter-process communication (IPC).
    - **Overhead**: Higher overhead due to context switching and memory management between processes.

```
public class MultitaskingDemo {
    public static void main(String[] args) {
        ProcessBuilder processBuilder = new
ProcessBuilder("notepad.exe");
        try {
            Process process = processBuilder.start();
            System.out.println("Notepad started.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Thread Life Cycle and Priorities

1. **Thread Life Cycle**:
    - **New**: A thread is in the new state if you create an instance of `Thread` class but before the invocation of `start()` method.

```
Thread thread = new Thread();
```

- **Runnable**: A thread is in the runnable state after invocation of `start()` method but before the thread is selected to run by the scheduler.

```
thread.start();
```

- **Running**: The thread is in running state if the thread scheduler has selected it.

```
public void run() {
    System.out.println("Thread is running");
}
```

- **Blocked (or Waiting)**: A thread is in a blocked state when it is waiting for a resource to become available or for another thread to perform a particular action.

```
synchronized (resource) {
    resource.wait();
}
```

- **Timed Waiting**: A thread that is waiting for a specified amount of time.

```
Thread.sleep(1000);  // Timed waiting for 1 second
```

- **Terminated (Dead)**: A thread is in the terminated or dead state when its run method exits.

```
thread.join();
```

```
public class ThreadLifecycleDemo extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }

    public static void main(String[] args) {
        ThreadLifecycleDemo thread = new ThreadLifecycleDemo();
        System.out.println("Thread state: " + thread.getState()); //
NEW
        thread.start();
        System.out.println("Thread state: " + thread.getState()); //
RUNNABLE
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread state: " + thread.getState()); //
TERMINATED
    }
}
```

2. **Thread Priorities**:
   - **Thread Priority**: Determines the relative priority of a thread.
   - **Default Priority**: By default, every thread is given priority `5`.
   - **Priority Constants**:
     - `MIN_PRIORITY` (1)
     - `NORM_PRIORITY` (5)
     - `MAX_PRIORITY` (10)

```
public class ThreadPriorityDemo extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " with
priority " + Thread.currentThread().getPriority() + " is running");
    }

    public static void main(String[] args) {
        ThreadPriorityDemo thread1 = new ThreadPriorityDemo();
        ThreadPriorityDemo thread2 = new ThreadPriorityDemo();
        ThreadPriorityDemo thread3 = new ThreadPriorityDemo();

        thread1.setPriority(Thread.MIN_PRIORITY);
        thread2.setPriority(Thread.NORM_PRIORITY);
        thread3.setPriority(Thread.MAX_PRIORITY);

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

Synchronizing Threads and Inter-Thread Communication

1. **Synchronizing Threads**:
   - **Purpose**: To prevent thread interference and memory consistency errors.
   - **Synchronized Method**: A method that can be accessed by only one thread at a time.

  - **Synchronized Block**: A block of code that can be executed by only one thread at a time.

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

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

    public int getCount() {
        return count;
    }
}

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

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Count: " + counter.getCount()); // Output:
2000
    }
}
```

2. **Inter-Thread Communication**:
  - **Purpose**: To allow threads to communicate with each other.
  - **Methods**:

- `wait()` : Causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` methods for this object.
- `notify()` : Wakes up a single thread that is waiting on this object's monitor.
- `notifyAll()` : Wakes up all threads that are waiting on this object's monitor.

```
class SharedResource {
    private int number;
    private boolean available = false;

    public synchronized int get() {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        available = false;
        notifyAll();
        return number;
    }

    public synchronized void put(int number) {
        while (available) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.number = number;
        available = true;
        notifyAll();
    }
}

public class InterThreadCommDemo {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.put(i);
                System.out.println("Produced: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                int value = resource.get();
                System.out.println("Consumed: " + value);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Thread Groups and Daemon Threads

1. **Thread Groups**:
   - **Purpose**: To manage multiple threads as a single unit.
   - **Creating Thread Groups**:
     - Use the `ThreadGroup` class to create a group.
     - Can specify a parent group; otherwise, the new group is added to the default group.

```
public class ThreadGroupDemo implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("Group A");

        Thread t1 = new Thread(group, new ThreadGroupDemo(), "Thread
1");
        Thread t2 = new Thread(group, new ThreadGroupDemo(), "Thread
2");
        Thread t3 = new Thread(group, new ThreadGroupDemo(), "Thread
3");
```

```
        t1.start();
        t2.start();
        t3.start();

        System.out.println("Thread Group Name: " + group.getName());
        group.list();
    }
}
```

- **Methods of** `ThreadGroup`:
  - `activeCount()`: Returns the number of active threads in the group.
  - `activeGroupCount()`: Returns the number of active groups in the group.
  - `list()`: Prints information about the thread group to the standard output.
  - `interrupt()`: Interrupts all threads in the group.

2. **Daemon Threads**:
   - **Purpose**: Daemon threads are service providers for other threads running in the same process.
   - **Characteristics**:
     - Runs in the background.
     - JVM terminates the daemon threads when all user threads (non-daemon threads) finish their execution.
     - Example: Garbage Collector.

```
public class DaemonThreadDemo extends Thread {
    public void run() {
        if (Thread.currentThread().isDaemon()) {
            System.out.println("Daemon thread is running");
        } else {
            System.out.println("User thread is running");
        }
    }

    public static void main(String[] args) {
        DaemonThreadDemo t1 = new DaemonThreadDemo();
        DaemonThreadDemo t2 = new DaemonThreadDemo();
        DaemonThreadDemo t3 = new DaemonThreadDemo();

        t1.setDaemon(true);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Enumerations, Autoboxing, Annotations, Generics

1. **Enumerations**:
    - **Purpose**: To define a set of named constants.
    - **Syntax**:

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumDemo {
    public static void main(String[] args) {
        Day day = Day.MONDAY;
        System.out.println("Day: " + day);

        for (Day d : Day.values()) {
            System.out.println(d);
        }
    }
}
```

2. **Autoboxing and Unboxing**:
    - **Autoboxing**: Automatic conversion of primitive types to their corresponding object wrapper classes.
    - **Unboxing**: Automatic conversion of wrapper classes to their corresponding primitive types.

```
public class AutoboxingUnboxingDemo {
    public static void main(String[] args) {
        // Autoboxing
        int a = 10;
        Integer aObj = a;

        // Unboxing
        Integer bObj = new Integer(20);
        int b = bObj;

        System.out.println("Autoboxing: " + aObj);
        System.out.println("Unboxing: " + b);
    }
}
```

3. **Annotations**:
   - **Purpose**: To provide metadata for Java code.
   - **Built-in Annotations**:
     - `@Override` : Indicates that a method overrides a method in a superclass.
     - `@Deprecated` : Marks a method as deprecated.
     - `@SuppressWarnings` : Suppresses specific warnings.
   - **Custom Annotations**: Define your own annotations.

```
// Custom Annotation
@interface MyAnnotation {
    String value();
}

public class AnnotationDemo {
    @MyAnnotation(value = "Hello")
    public void myMethod() {
        System.out.println("Annotated method");
    }

    public static void main(String[] args) {
        AnnotationDemo demo = new AnnotationDemo();
        demo.myMethod();
    }
}
```

4. **Generics**:
   - **Purpose**: To enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.
   - **Syntax**: Use angle brackets to specify the type parameter.

```
public class GenericClass<T> {
    private T obj;

    public void set(T obj) {
        this.obj = obj;
    }

    public T get() {
        return obj;
    }

    public static void main(String[] args) {
        GenericClass<Integer> intObj = new GenericClass<>();
        intObj.set(10);
        System.out.println("Integer Value: " + intObj.get());

        GenericClass<String> strObj = new GenericClass<>();
        strObj.set("Hello");
        System.out.println("String Value: " + strObj.get());
    }
}
```

- **Generic Methods**:

```
public class GenericMethodDemo {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        String[] strArray = {"A", "B", "C"};

        printArray(intArray);
        printArray(strArray);
    }
}
```