MOD 5

Q1) Explain try, catch, throw, throws, finally keywords with examples

**1. try**

The try block contains code that might throw an exception. It helps in identifying and isolating code that could lead to an error.

**Example:**

Java

try {

int result = 10 / 0; // This will throw an ArithmeticException

System.out.println(result);

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

}

**2. catch**

The catch block handles the exception thrown in the try block. You can have multiple catch blocks to handle different exceptions.

**Example:**

Java

try {

int[] arr = {1, 2, 3};

System.out.println(arr[5]); // ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index is out of bounds!");

} catch (Exception e) {

System.out.println("An error occurred: " + e);

}

**3. throw**

The throw keyword is used to explicitly throw an exception, either predefined or user-defined.

**Example:**

Java

public void checkAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or older.");

} else {

System.out.println("Access granted.");

}

}

public static void main(String[] args) {

try {

checkAge(16);

} catch (IllegalArgumentException e) {

System.out.println(e.getMessage());

}

}

**4. throws**

The throws keyword declares the exceptions a method can throw. This helps the caller know what to handle.

**Example:**

Java

public void readFile() throws IOException {

FileReader file = new FileReader("nonexistent.txt");

BufferedReader fileInput = new BufferedReader(file);

System.out.println(fileInput.readLine());

}

public static void main(String[] args) {

try {

readFile();

} catch (IOException e) {

System.out.println("File not found!");

}

}

**5. finally**

The finally block contains code that executes after the try or catch block, regardless of whether an exception occurred. It's often used for cleanup operations like closing files or releasing resources.

**Example:**

Java

try {

int result = 10 / 2;

System.out.println(result);

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero.");

} finally {

System.out.println("Execution complete.");

}

**Combined Example:**

Java

public static void main(String[] args) {

try {

int num = Integer.parseInt("XYZ"); // This will throw NumberFormatException

} catch (NumberFormatException e) {

System.out.println("Invalid number format: " + e.getMessage());

throw new RuntimeException("Re-throwing exception.");

} finally {

System.out.println("Cleanup actions completed.");

}

}

**Summary:**

* **try:** Encloses code that may cause an exception.
* **catch:** Handles the exception.
* **throw:** Used to explicitly throw an exception.
* **throws:** Declares the exceptions a method may throw.
* **finally:** Always executes, useful for cleanup.

Q2) Checked vs Unchecked Exceptions

**1. Checked Exception**

Checked exceptions are exceptions that must be either **declared** in the throws clause of a method or **caught** in a try-catch block. The compiler checks these at compile time.

**Example Code:**

Java

import java.io.\*;

public class CheckedExample {

public static void readFile() throws IOException {

FileReader file = new FileReader("file.txt");

BufferedReader reader = new BufferedReader(file);

System.out.println(reader.readLine());

}

public static void main(String[] args) {

try {

readFile(); // Method declares IOException

} catch (IOException e) {

System.out.println("File not found or cannot be read: " + e.getMessage());

}

}

}

**2. Unchecked Exception**

Unchecked exceptions are exceptions that the compiler does NOT require you to handle or declare. These occur due to programming errors or logical flaws.

**Examples of Unchecked Exceptions:**

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

**Example Code:**

Java

public class UncheckedExample {

public static void divideByZero() {

int result = 10 / 0; // This will throw ArithmeticException

}

public static void main(String[] args) {

try {

divideByZero();

} catch (ArithmeticException e) {

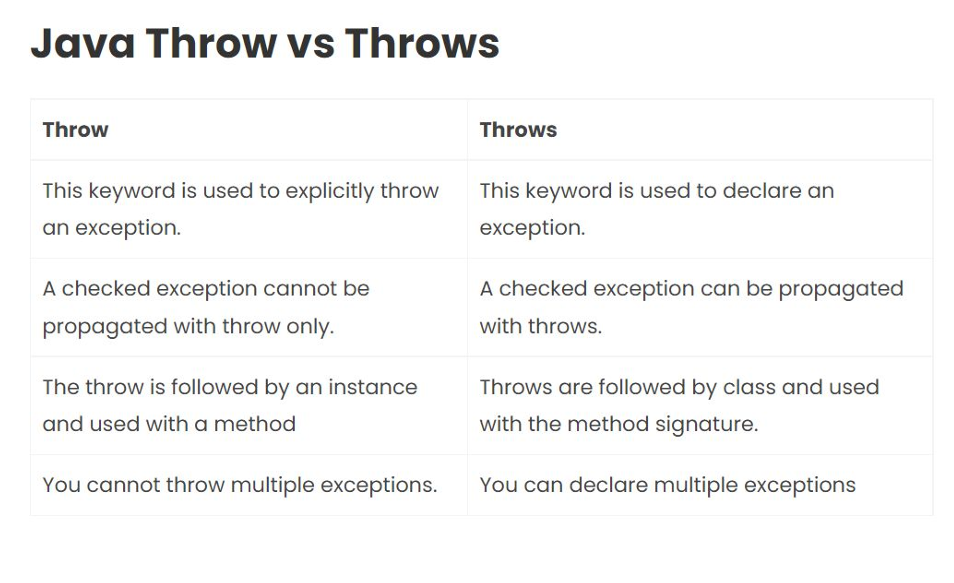
System.out.println("Cannot divide by zero: " + e.getMessage());

}

}

}

Q3) throw vs throws



Q4) What are user defined exception?

User-defined exceptions are custom exceptions created by extending the **Exception** class (for checked exceptions) or the **RuntimeException** class (for unchecked exceptions). These are used when predefined exceptions do not suffice for a specific error or condition in your program.

Let's create a custom exception to validate if a user is old enough to vote.

Java

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message); // Pass the message to the superclass constructor

}

}

public class UserDefinedExceptionExample {

// Method to validate age

public static void checkAge(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or older to vote.");

} else {

System.out.println("You are eligible to vote.");

}

}

public static void main(String[] args) {

try {

checkAge(16); // This will throw the custom exception

} catch (InvalidAgeException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

Q5) Exception handling mechanism

1. **Definition of Exception**:

* An exception is an event that disrupts the normal flow of a program during runtime.

2. **Key Components**:

* **try Block**: Contains code that might throw an exception.
* **catch Block**: Handles the exception thrown in the try block.
* **finally Block**: Executes code regardless of whether an exception was thrown or caught.
* **throw Statement**: Used to explicitly throw an exception.
* **throws Keyword**: Declares exceptions a method might throw.

3. **Flow of Control**:

* The program enters the try block.
* If no exception occurs, the try block completes, and the program skips the catch block.
* If an exception occurs, control immediately transfers to the first catch block that matches the exception type.
* The finally block always executes after the try-catch block, regardless of whether an exception was thrown or handled.

Java

try {

// Code that may throw an exception

} catch (ExceptionType e) {

// Code to handle the exception

} finally {

// Code that always executes

}

// Throwing an exception

throw new ExceptionType("Error message");

// Declaring exceptions in a method

public void method() throws ExceptionType {

// Method code

}

Q6) Benefits of handling exceptions explicitly

1. **Prevents Program Crashes**

* Ensures the program continues running even after encountering errors.

2. **Improves Code Readability and Maintainability**

* Separates error-handling logic from regular code, making it more structured.

3. **Allows Recovery from Errors**

* Enables you to recover from runtime errors gracefully (e.g., retrying operations).

4. **Provides Meaningful Error Messages**

* Helps users and developers understand what went wrong.

5. **Improves User Experience**

* Improves UX.

Q7) Packages

**Access Levels for Packages**

| **Access Modifier** |  |  | **Same Class** |  | **Same Package** |  | **Subclass (Different Package)** |  |  | **World** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **public** |  |  | ✅ |  | ✅ |  | ✅ |  |  | ✅ |
| **protected** |  |  | ✅ |  | ✅ |  | ✅ |  |  | ❌ |
| **default** (no modifier) |  |  | ✅ |  | ✅ |  | ❌ |  |  | ❌ |
| **private** |  |  | ✅ |  | ❌ |  | ❌ |  |  | ❌ |

**User-Defined Packages**

**Definition**: A **user-defined package** is a custom package created by a developer to group related classes and interfaces.

**Benefits**:

1. **Organized Code**: Keeps code modular and manageable.
2. **Reusability**: Allows reuse of classes across projects.
3. **Namespace Management**: Prevents naming conflicts.
4. **Access Control**: Provides controlled access via access modifiers.

**Syntax**:

1. **Creating a Package**:

Java

package mypackage; // Declare at the top of the file

public class MyClass {

public void display() {

System.out.println("Hello from MyClass!");

}

}

1. **Using a Package**:

java

import mypackage.MyClass; // Import the package

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.display();

}

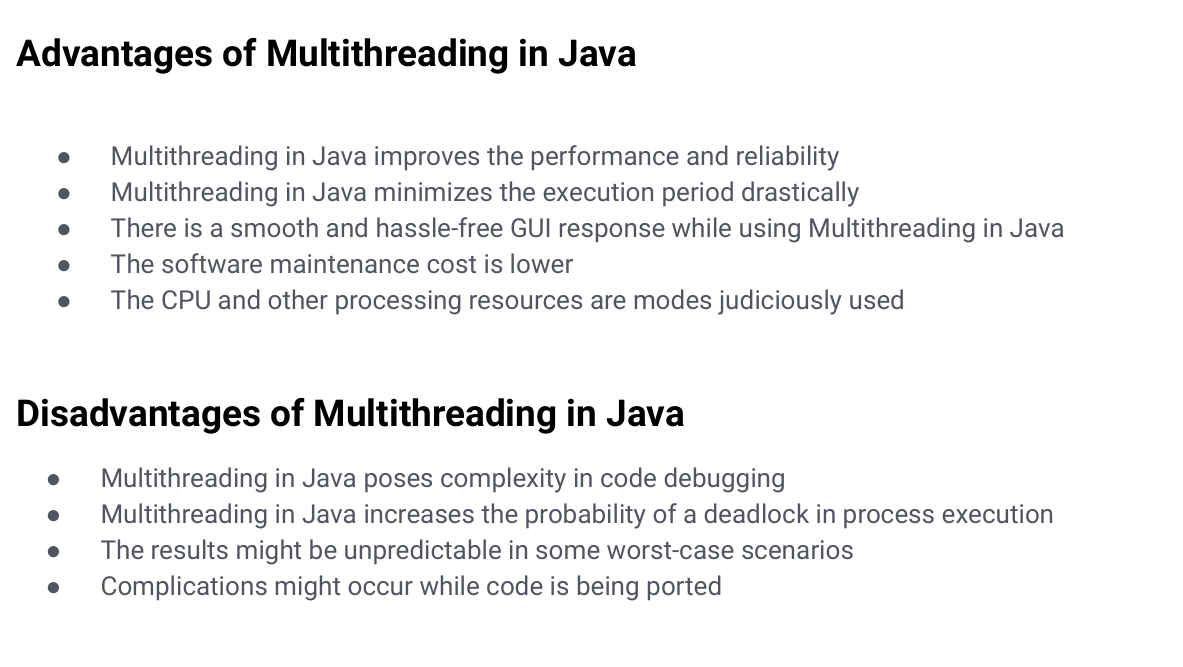
}

**Steps to Create and Use User-Defined Packages**

1. **Create a Package**:
   * Use the package keyword at the top of your class file.
   * Save the file in a folder with the package name.
2. **Compile the Package**:
   * Navigate to the folder where the class file is saved.
   * Use javac -d . MyClass.java to compile and create the package directory.
3. **Use the Package in Another Program**:
   * Use the import keyword to include the package in another program.
   * Instantiate and use the class from the package.

Q8) Multithreading

Multithreading is the process of executing multiple threads simultaneously within a single program to perform tasks concurrently, enhancing the efficiency of CPU usage.



Can be achieved by using

1. Extending Thread Class
2. Runnable Interface

1)

class MyThread extends Thread{

@Override

public void run(){

int i =0;

while(i<40000){

System.out.println("My Cooking Thread is Running");

System.out.println("I am happy!");

i++;

}

}

}

public class cwh\_70 {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

}

}

2)

classs t1 implements Runnable{

@Override

public void run(){

System.out.println("Thread is running");

}

}

public class ClassName{

public static void main(String[] args) {

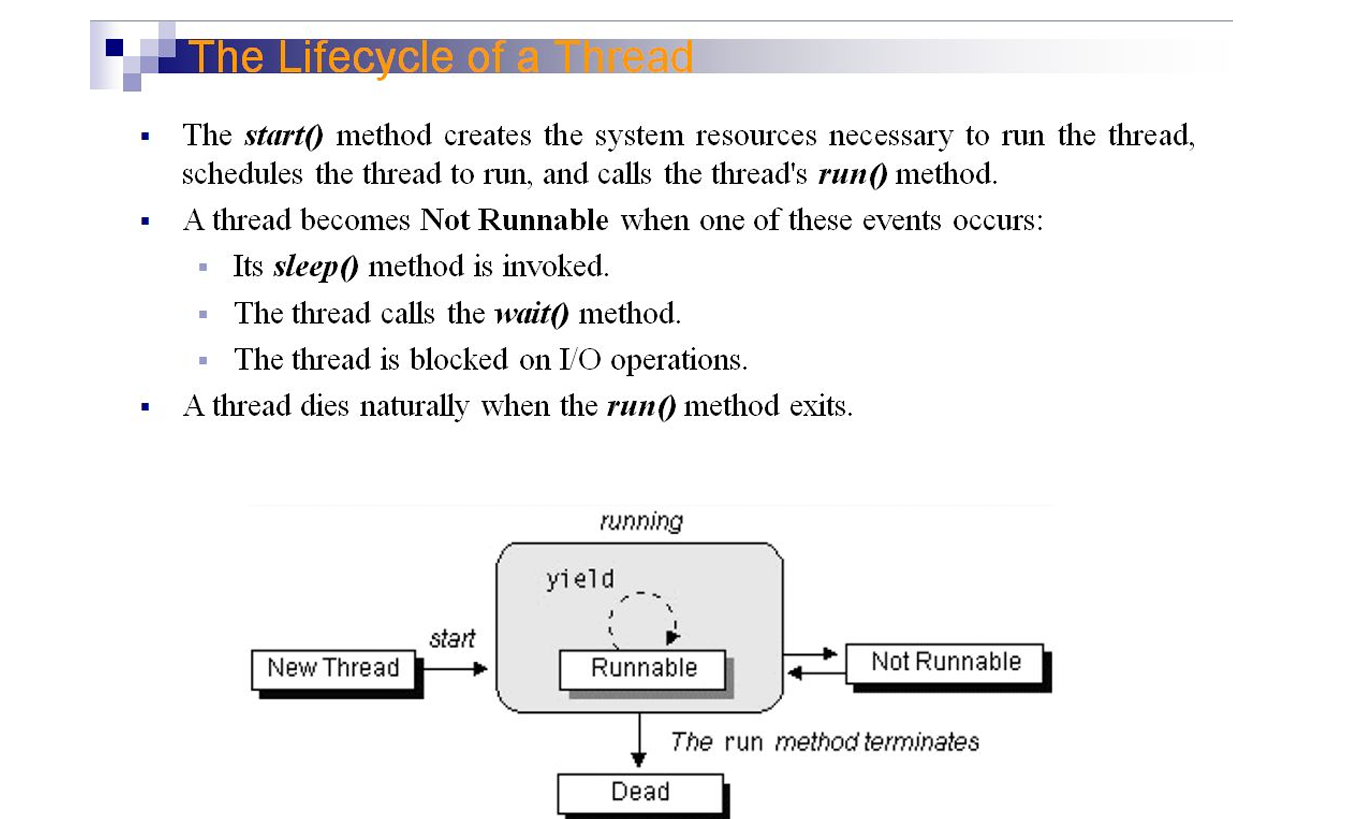
t1 obj1 = new t1();

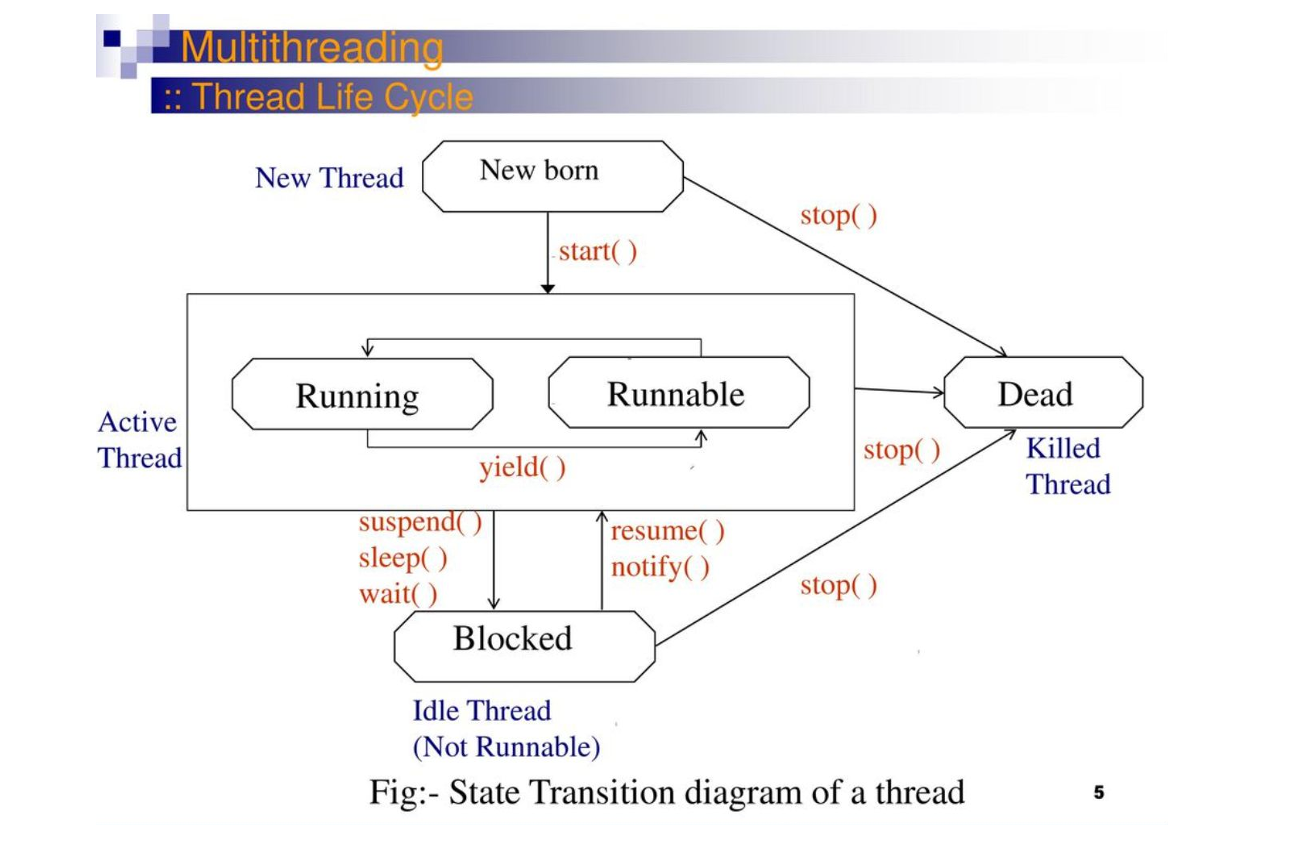
Thread t = new Thread(obj1);

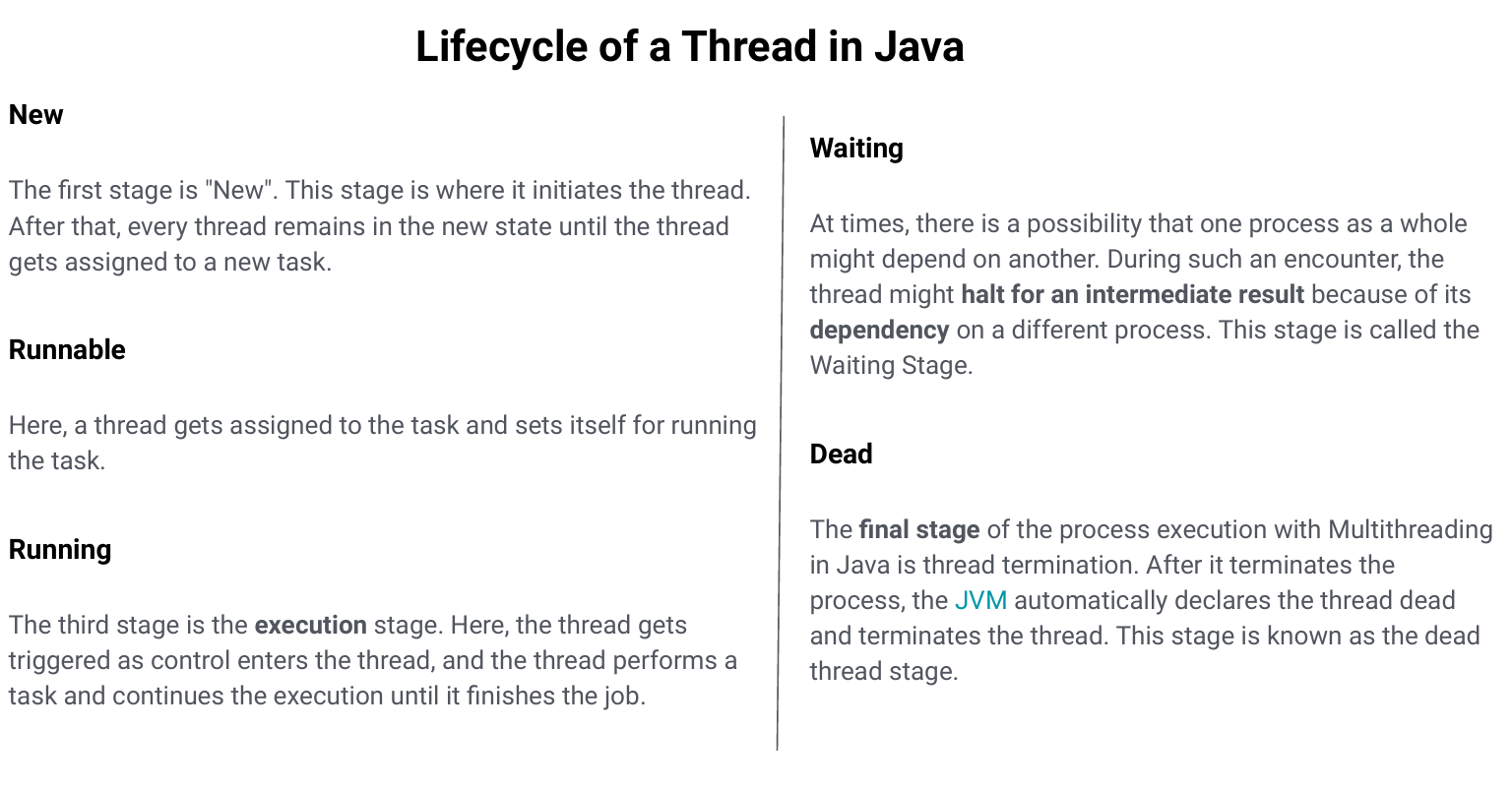
t.start();

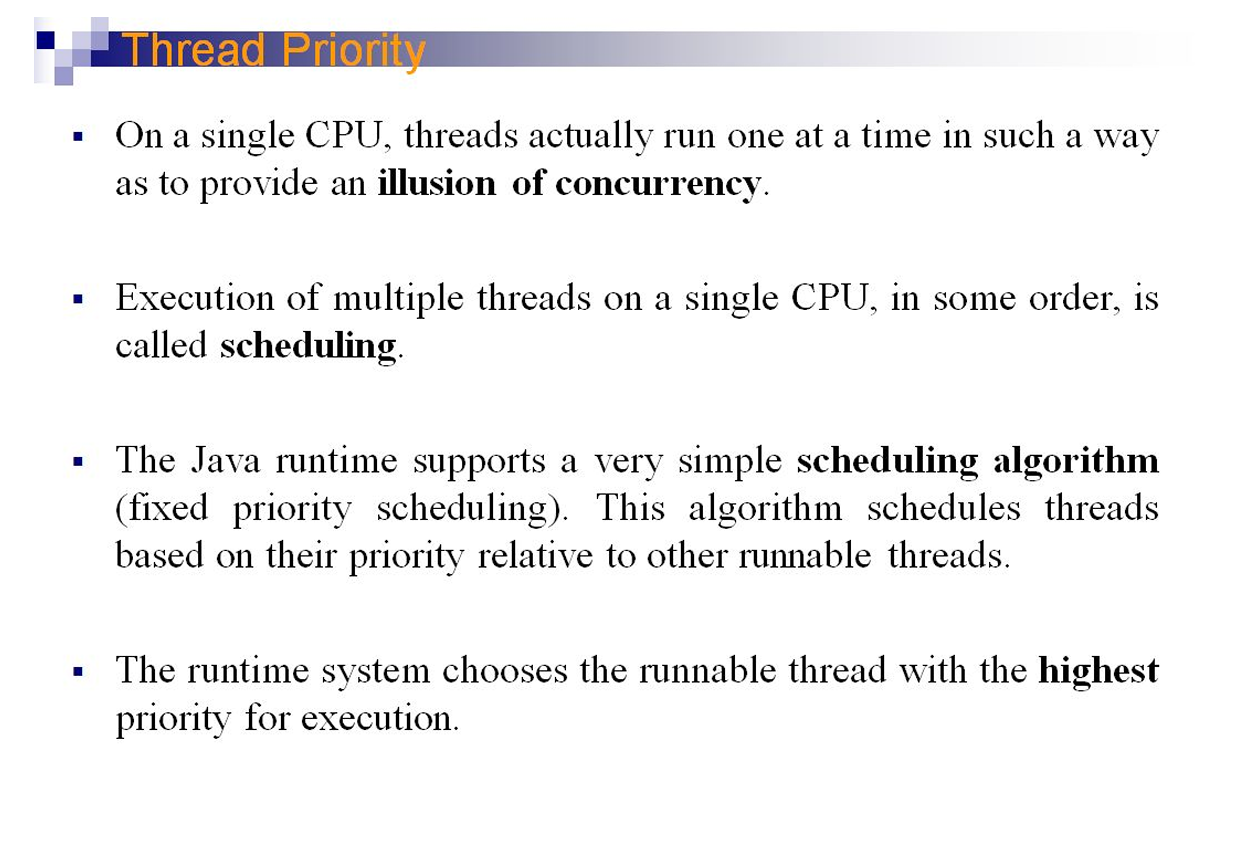
}}

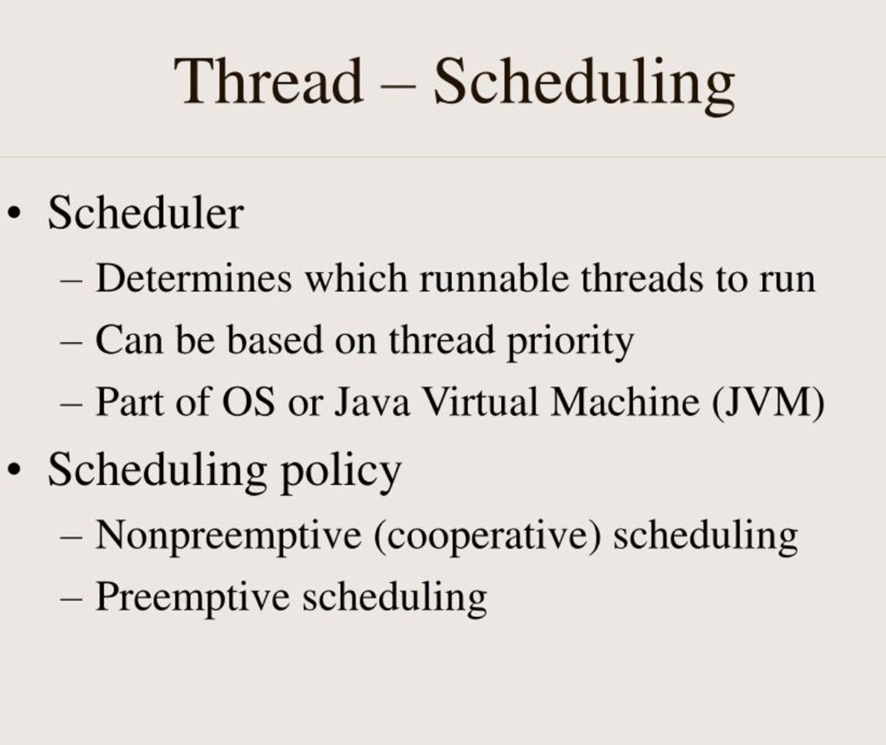
Q9) Lifecycle of thread & Stages

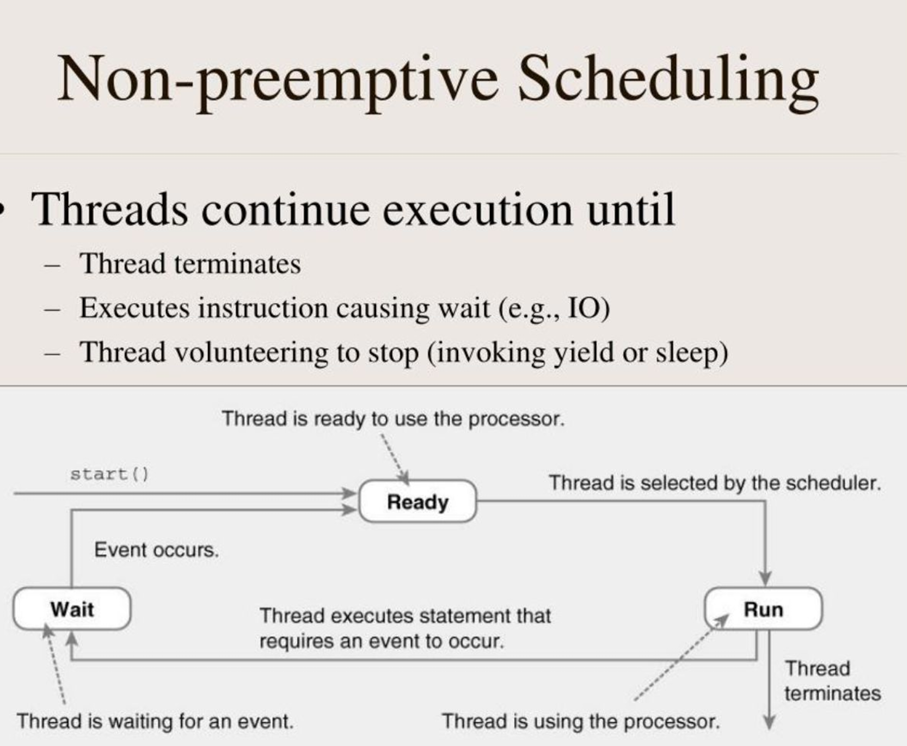


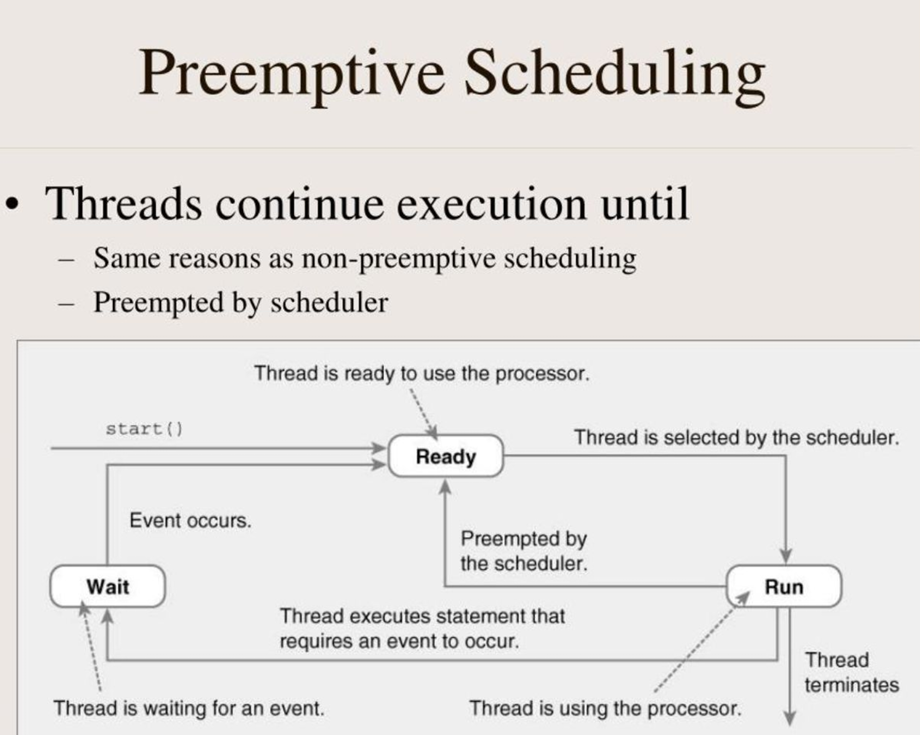




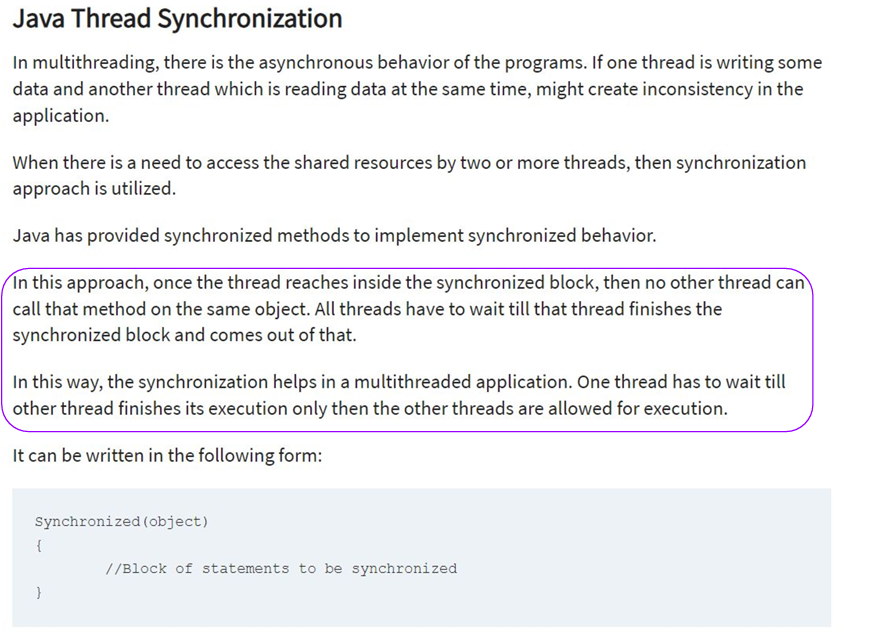


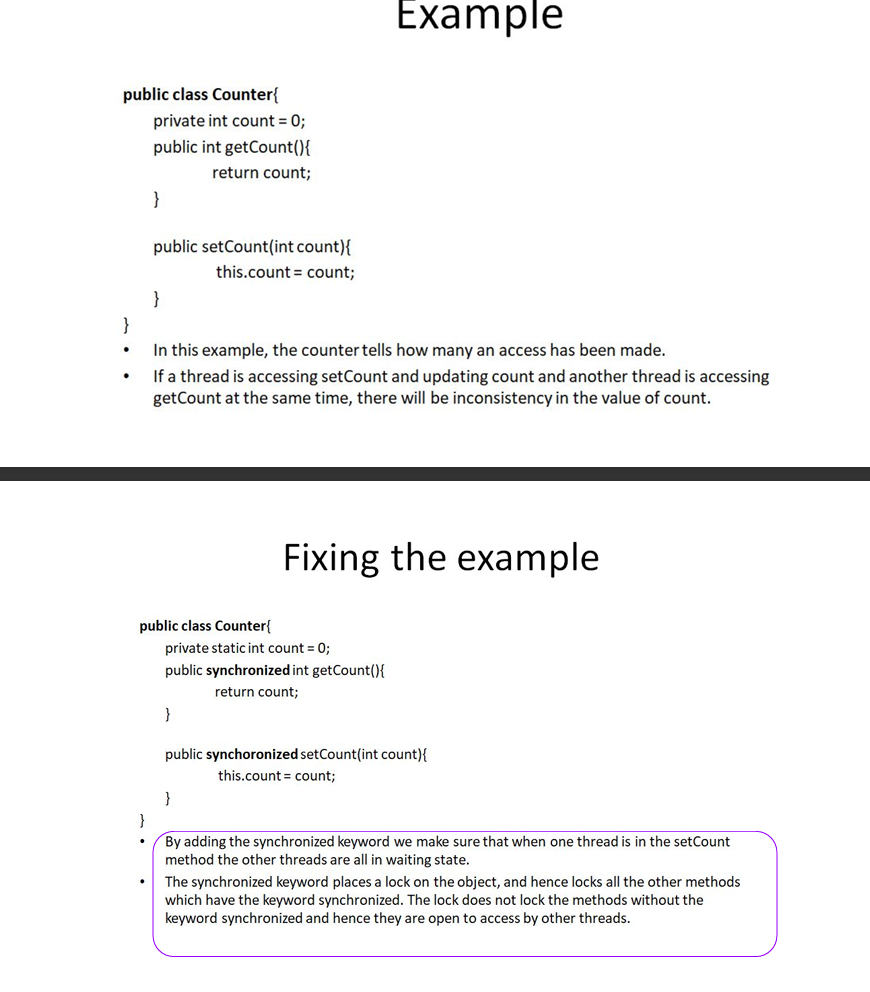






Synchronization





Deadlocks

