**Threading in Java**

Threading is a fundamental concept in Java that allows multiple threads (lightweight units of a process) to run concurrently, sharing CPU time.

This is essential for creating efficient, high-performance applications, particularly when dealing with tasks that can be performed simultaneously, like downloading files, processing large data sets, or handling multiple users in a web server.

**Key Concepts**

1. **Thread**: A thread is the smallest unit of execution in a process. Java supports multithreading by using the Thread class and the Runnable interface.
2. **Multithreading**: This is the ability to execute multiple threads simultaneously. Java's multithreading model is built on top of the native OS threads, meaning the JVM uses the OS-level threads.
3. **Concurrency vs. Parallelism**:
   * **Concurrency** refers to multiple tasks making progress simultaneously (can happen on a single core by switching between tasks).
   * **Parallelism** refers to multiple tasks executing at the exact same time, which typically requires multiple CPU cores.

**Process:**

In the context of threading, a **process** is an independent program in execution, which has its own memory space, code, data, and system resources.

It serves as the container within which threads are created and executed.

Here's a detailed explanation:

**Key Concepts of a Process in Threading:**

1. **Memory Space**:
   * A process has its own separate memory space, which includes the code segment, data segment (variables, constants), and stack.
   * This memory isolation ensures that processes do not interfere with each other directly.
   * Threads within the same process share the process's memory space, allowing them to communicate and share data more easily.
2. **Resources**:
   * Each process is assigned resources by the operating system, including CPU time, file handles, network connections, and access to other system resources.
   * Threads within the same process share these resources.
3. **Execution Context**:
   * A process includes an execution context, which comprises the process's state, including its registers, program counter, and stack pointer.
   * When a process is switched out for another, this context is saved so the process can resume correctly later.

**Processes vs. Threads:**

* **Process**:
  + A process is an independent entity, with its own memory space and resources.
  + Inter-process communication (IPC) is required for processes to communicate with each other, which can be more complex and slower compared to thread communication.
  + Creating a new process is resource-intensive as it requires allocating separate memory and resources.
* **Thread**:
  + A thread is the smallest unit of execution within a process.
  + Threads within the same process share the same memory space, making inter-thread communication more efficient.
  + Creating a new thread within the same process is less resource-intensive because it doesn't require a new memory space.

**Importance of Processes in Threading:**

* **Isolation**: Processes are isolated from each other, which enhances security and stability. If one process crashes, it doesn't affect other processes.
* **Concurrency**: Multiple processes can run concurrently on separate CPU cores. Within each process, multiple threads can also run concurrently.
* **Multi-threading**: Within a process, threads allow concurrent execution of tasks, leading to more efficient utilization of CPU resources.

**Example:**

When you run a web browser (like Chrome or Firefox), the browser itself is a process. It may use multiple threads to handle different tasks like rendering the web page, downloading resources, and handling user input simultaneously. Each tab might run in its own process to ensure that if one tab crashes, it doesn't affect the others.

**Creating Threads in Java**

There are two main ways to create a thread:

1. **Extending the Thread class**
2. **Implementing the Runnable interface**

**1. Extending the Thread Class**

**Code Example: Restaurant Kitchen**

In this version, each chef starts preparing their dish, and the main thread continues without waiting for the chefs to finish.

This is akin to a scenario where dishes are being prepared, and the restaurant manager is free to perform other tasks while the chefs are working.

// Class that simulates a Chef preparing a dish in a separate thread

class **Chef** extends **Thread** {

private String dishName;

private int preparationTime;

// Constructor to pass the name of the dish and its preparation time

public Chef(String dishName, int preparationTime) {

this.dishName = dishName;

this.preparationTime = preparationTime;

}

// Overriding the run method to simulate dish preparation

@Override

public void run() {

System.out.println("Chef started preparing " + dishName);

try {

// Simulate time taken to prepare the dish

Thread.sleep(preparationTime);

} catch (InterruptedException e) {

System.out.println("Preparation interrupted for " + dishName);

return;

}

System.out.println("Chef finished preparing " + dishName);

}

}

public class **Restaurant** {

public static void main(String[] args) {

// Create multiple threads for different dishes

Chef dish1 = new Chef("Pasta", 2000); // Preparation time: 2 seconds

Chef dish2 = new Chef("Pizza", 3000); // Preparation time: 3 seconds

Chef dish3 = new Chef("Salad", 1500); // Preparation time: 1.5 seconds

// Start all the chef threads

dish1.start();

dish2.start();

dish3.start();

// Main thread does not wait for chefs to finish

System.out.println("\nMain thread is free to perform other tasks while chefs are working.\n");

}

}

**Output**

Main thread is free to perform other tasks while chefs are working.

Chef started preparing Pasta

Chef started preparing Pizza

Chef started preparing Salad

Chef finished preparing Salad

Chef finished preparing Pasta

Chef finished preparing Pizza

**Explanation**

* **Independent Execution**:
  + Each Chef (thread) works independently to prepare their dish.
  + The start() method begins the execution of each thread, and they run concurrently without the main thread waiting for their completion.
* **Concurrency in Action**:
  + The chefs (threads) work concurrently, demonstrating how multiple tasks can progress simultaneously without blocking the main program's flow.

**Real-World Application**

This example simulates situations where tasks are delegated and can run independently without requiring the main process to wait for their completion.

Examples include:

* A server handling multiple client requests concurrently.
* Background tasks (like sending emails) in an application while the main process continues handling user input.
* Running independent processes in a GUI application without freezing the interface.

This approach is useful when you want to offload work to separate threads without caring when they finish, allowing the main process to continue with other tasks.

1. **Implementing the Runnable Interface**

Here's how you can rewrite the restaurant kitchen example using the Runnable interface.

In this version, each chef is represented by a Runnable task that simulates preparing a dish.

Using Runnable is beneficial when you want to decouple the task to be performed from the thread itself, allowing for more flexibility and reusability of the task.

**Code Example: Restaurant Kitchen Using Runnable**

// Class that simulates a Chef preparing a dish using the Runnable interface

class **Chef** implements **Runnable** {

private String dishName;

private int preparationTime;

// Constructor to pass the name of the dish and its preparation time

public Chef(String dishName, int preparationTime) {

this.dishName = dishName;

this.preparationTime = preparationTime;

}

// Overriding the run method to simulate dish preparation

@Override

public void run() {

System.out.println("Chef started preparing " + dishName);

try {

// Simulate time taken to prepare the dish

Thread.sleep(preparationTime);

} catch (InterruptedException e) {

System.out.println("Preparation interrupted for " + dishName);

return;

}

System.out.println("Chef finished preparing " + dishName);

}

}

public class **Restaurant** {

public static void main(String[] args) {

// Create Runnable tasks for different dishes

Runnable dish1 = new Chef("Pasta", 2000); // Preparation time: 2 seconds

Runnable dish2 = new Chef("Pizza", 3000); // Preparation time: 3 seconds

Runnable dish3 = new Chef("Salad", 1500); // Preparation time: 1.5 seconds

// Create threads and pass the Runnable tasks to them

Thread chef1 = new Thread(dish1);

Thread chef2 = new Thread(dish2);

Thread chef3 = new Thread(dish3);

// Start all the chef threads

chef1.start();

chef2.start();

chef3.start();

// Main thread does not wait for chefs to finish

System.out.println("\nMain thread is free to perform other tasks while chefs are working.\n");

}

}

**Output**

Chef started preparing Pasta

Chef started preparing Pizza

Chef started preparing Salad

Main thread is free to perform other tasks while chefs are working.

Chef finished preparing Salad

Chef finished preparing Pasta

Chef finished preparing Pizza

**Explanation**

* **Using Runnable**:
  + By implementing the Runnable interface, we separate the task (preparing a dish) from the thread itself.
  + This makes the Chef class more flexible and allows it to be used with different thread mechanisms.
* **Creating Threads**:
  + We create new Thread objects and pass instances of Chef (which implement Runnable) to the thread constructor.
* **Starting Threads**:
  + We call start() on each Thread object to begin the execution of the run() method in parallel.
* **Concurrent Execution**:
  + Like before, each thread works independently, and the main thread does not wait for them to finish, showcasing how multiple tasks can run concurrently.

**Advantages of Using Runnable**

* **Separation of Concerns**:
  + The Runnable interface allows the task to be separated from the thread itself, promoting better design.
* **Flexibility**:
  + Since Runnable can be passed to any thread, it's more versatile and can be used with thread pools and executors.
* **Reusability**:
  + The same Runnable instance can be reused across multiple threads if needed.

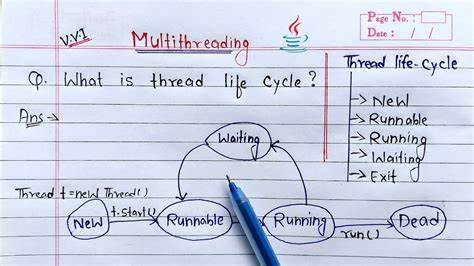
**Real-World Application**

Using Runnable is common when you want to execute a task in a separate thread without extending the Thread class, allowing the class to extend other classes or be used in a more flexible context.

This is useful in scenarios like:

* Running background tasks in a web server.
* Executing concurrent processes in a GUI application.
* Handling concurrent data processing in a multi-threaded environment.

**Thread Lifecycle**



A thread goes through various states during its lifecycle:

1. **New**: When a thread is created but not yet started (new Thread()).
2. **Runnable**: After calling start(), the thread is ready to run but may not be running immediately, depending on the CPU scheduler.
3. **Running**: When the thread is executing its run() method.
4. **Waiting(Blocked)**: When the thread is waiting for a resource or another thread.
5. **Dead(Terminated)**: When the thread has finished execution (run() method ends).

Threads in programming go through several phases during their lifecycle.

Here's a detailed explanation of each phase:

**1. New**

* **Description**: This is the initial state of a thread when it is created but not yet started. In this state, the thread is not scheduled for running.
* **Example**: When you create a thread using new Thread(), it enters the New state.
* **Characteristics**:
  + The thread is not yet considered alive.
  + Resources for the thread are allocated but not yet utilized.

**2. Runnable**

* **Description**: After invoking the start() method on a thread, it enters the Runnable state. In this state, the thread is ready to run and is waiting for the CPU to assign it execution time.
* **Example**: After calling thread.start(), the thread enters this state.
* **Characteristics**:
  + The thread is in a queue waiting for CPU time.
  + It can transition between Runnable and Running depending on CPU availability.

**3. Running**

* **Description**: This state occurs when the thread is actually executing its run() method. The CPU scheduler picks the thread from the Runnable state and assigns it CPU time.
* **Example**: When the thread's run() method is being executed, the thread is in the Running state.
* **Characteristics**:
  + The thread actively uses CPU resources.
  + It can move back to Runnable or into a Waiting state if it needs to wait for resources or if it's preempted.

**4. Waiting (Blocked)**

* **Description**: A thread enters the Waiting or Blocked state when it needs to wait for another thread to perform a task, or when it is waiting for a resource such as I/O or a lock.
* **Example**: When a thread calls wait() or is trying to acquire a lock that is held by another thread.
* **Characteristics**:
  + The thread is not eligible for CPU time.
  + It remains in this state until the resource becomes available or the other thread completes its task.
  + Includes sub-states like TIMED\_WAITING (waiting for a specified amount of time) and BLOCKED (waiting for a monitor lock).

**5. Dead (Terminated)**

* **Description**: The thread enters the Dead or Terminated state when it has completed execution. This happens either when the run() method finishes or when the thread is explicitly terminated.
* **Example**: When the thread’s run() method exits normally or throws an unhandled exception.
* **Characteristics**:
  + The thread cannot be restarted.
  + All the resources used by the thread are released.

Each of these phases represents a step in the lifecycle of a thread, starting from creation to termination, and determines how the thread interacts with the system's CPU scheduler and other threads.

**Thread Methods**

* **start()**: Starts a thread by invoking the run() method.
* **sleep(milliseconds)**: Pauses the execution of the thread for a specified time.
* **join()**: Allows one thread to wait for the completion of another.
* **interrupt()**: Signals a thread to stop its current action.

**Synchronization**

When multiple threads access shared resources, there’s a risk of data inconsistency (race conditions). To prevent this, **synchronization** is used to allow only one thread to access the resource at a time.

public class **Counter** {

private int count = 0;

// Synchronized method to ensure only one thread can execute it at a time

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class Main {

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());

}

}

* **Explanation**:
  + The increment() method is synchronized, meaning only one thread can access it at a time.
  + Without synchronization, both threads might try to modify the count simultaneously, causing incorrect results.

**Deadlock**

Deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource.

**Example of potential deadlock:**

class **Deadlock** {

private final Object lock1 = new Object();

private final Object lock2 = new Object();

public void method1() {

synchronized (lock1) {

System.out.println("Lock1 acquired, waiting for Lock2");

synchronized (lock2) {

System.out.println("Method1: Both locks acquired");

}

}

}

public void method2() {

synchronized (lock2) {

System.out.println("Lock2 acquired, waiting for Lock1");

synchronized (lock1) {

System.out.println("Method2: Both locks acquired");

}

}

}

}

public class Main {

public static void main(String[] args) {

Deadlock deadlock = new Deadlock();

Thread t1 = new Thread(deadlock::method1);

Thread t2 = new Thread(deadlock::method2);

t1.start();

t2.start();

}

}

**Explanation**:

* method1() locks lock1 and waits for lock2, while method2() locks lock2 and waits for lock1.
* This situation leads to a deadlock, where neither thread can proceed because they are both waiting for resources held by the other.

**Advantages of Multithreading**

1. **Resource Sharing**: Threads share the same memory and resources, which leads to efficient utilization.
2. **Improved Performance**: Tasks like I/O operations can be performed in parallel, improving performance.
3. **Responsiveness**: Threads can make an application more responsive (e.g., keeping the UI responsive while performing background tasks).

**Disadvantages of Multithreading**

1. **Complexity**: Managing multiple threads, ensuring synchronization, and avoiding deadlocks can make programs more complex.
2. **Overhead**: Context switching between threads incurs overhead, which might reduce performance in certain cases.

Threading in Java is a powerful tool for parallelizing tasks but requires careful management of shared resources and thread coordination to avoid issues like race conditions or deadlocks.

**What are the different concepts in Threads?**

Threads in Java involve several key concepts that help in understanding how multithreading works and how to control the behavior of threads.

Here are the main concepts in threads:

**1. Thread Lifecycle (States of a Thread)**

A thread in Java passes through several stages from its creation to termination, known as the thread lifecycle.

These states are:

* **New**: A thread is in this state when it is created but not yet started (new Thread()).
* **Runnable**: After the start() method is called, the thread enters the runnable state. It is ready to run, but the actual running is controlled by the thread scheduler.
* **Running**: When the thread is executing its run() method, it is in the running state.
* **Blocked**: The thread is temporarily inactive because it is waiting for some resources (like I/O) or for a lock (in case of synchronized blocks).
* **Waiting**: A thread is waiting indefinitely for another thread to perform a particular action (e.g., using Object.wait() or Thread.join()).
* **Timed Waiting**: The thread is waiting for a specific period of time (e.g., using Thread.sleep() or Object.wait(timeout)).
* **Terminated**: Once the thread completes its execution or is terminated due to an exception, it enters the terminated state.

**2. Thread Priority**

Java threads have priorities that determine how the operating system's thread scheduler allocates CPU time.

A thread's priority is an integer between **MIN**\_**PRIORITY** (1) and **MAX**\_**PRIORITY** (10), with the default being **NORM**\_**PRIORITY** (5).

* The priority can be set using the **setPriority**() method, but the effect of priority can vary across different operating systems.

**3. Daemon vs User Threads**

* **User Threads**: These are normal threads that keep the JVM alive as long as any user thread is running. The program will not terminate until all user threads finish executing.
* **Daemon Threads**: These are background threads that run in the background to perform low-priority tasks. The JVM will terminate if only daemon threads are left running. Daemon threads are often used for services like garbage collection.
* To make a thread a daemon, use the **setDaemon**(true) method before starting the thread.

**4. Thread Synchronization**

**Synchronization** ensures that shared resources are accessed by only one thread at a time to avoid race conditions and maintain consistency.

* **Synchronized Methods**: A method can be marked as synchronized, which ensures that only one thread can execute it at a time.
* **Synchronized Blocks**: You can synchronize a block of code to lock on a specific object, allowing finer control over synchronization than synchronizing an entire method.
* **Intrinsic Lock (Monitor Lock)**: When a thread enters a synchronized method/block, it obtains a lock on the object or class, and other threads cannot enter synchronized methods of the same object/class until the lock is released.

**public synchronized void myMethod() {**

// Only one thread can access this method at a time

**}**

**5. Thread Communication (Inter-Thread Communication)**

Java provides mechanisms for threads to communicate with each other, especially when working with shared resources:

* **wait()**: A thread can call wait() to release the lock and go into a waiting state until another thread invokes notify() or notifyAll().
* **notify()**: Wakes up one waiting thread that is waiting on the object's monitor.
* **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

**6. Thread Pools**

Thread pools are used to manage a group of reusable threads, which can be assigned tasks, rather than creating new threads for each task.

This improves performance by reducing the overhead of thread creation.

* The Executor framework in Java (java.util.concurrent.**ExecutorService**) provides facilities for managing thread pools.

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.submit(new Task());

executor.shutdown();

**7. Deadlock**

Deadlock is a situation where two or more threads are blocked forever, each waiting for the other to release a resource.

For example, Thread 1 holds lock A and waits for lock B, while Thread 2 holds lock B and waits for lock A. This results in a situation where neither thread can proceed.

**8. Race Condition**

A race condition occurs when two or more threads try to modify shared data concurrently, leading to unpredictable or incorrect behavior.

Race conditions can be avoided by using synchronization to control access to shared resources.

**9. Thread Safety**

Thread safety ensures that shared data is accessed and modified correctly by multiple threads.

It can be achieved through synchronization, using atomic variables (AtomicInteger, AtomicReference, etc.), or concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList, etc.).

**10. Thread Scheduling**

The Java thread scheduler is part of the Java Virtual Machine (JVM) and determines which runnable thread will run.

It does this using:

* **Time-Slicing**: The scheduler allocates CPU time to threads in small time intervals, switching between threads.
* **Preemption**: The scheduler preempts a running thread if a higher-priority thread becomes runnable.

**11. Thread Join**

The join() method allows one thread to wait for the completion of another thread. This is useful when you want to ensure that a thread has completed its task before proceeding further in the program.

Thread t1 = new Thread(() -> {

// Task

});

t1.start();

t1.join(); // Main thread waits for t1 to finish before proceeding

**12. Thread Interruption**

Threads can be interrupted using the interrupt() method, which signals to the thread that it should stop what it's doing.

However, the thread must handle the interruption properly, as calling interrupt() does not stop the thread automatically.

Threads usually check for interruption status by calling **Thread.interrupted()** or catching **InterruptedException**.

Thread t1 = new Thread(() -> {

try {

Thread.sleep(10000);

}

catch (InterruptedException e) {

System.out.println("Thread interrupted");

}

});

t1.start();

t1.interrupt(); // Interrupts the thread

**13. Callable and Future**

In addition to Runnable, Java provides the Callable interface, which is similar to Runnable but allows the task to return a result or throw an exception. The result can be retrieved using a Future.

Callable<Integer> task = () -> {

return 42;

};

Future<Integer> future = executor.submit(task);

Integer result = future.get(); // Retrieves the result

**14. Volatile Keyword**

The volatile keyword ensures that the value of a variable is always read from the main memory and not from the thread's local cache.

This is used in cases where multiple threads access a shared variable and helps in avoiding memory consistency issues.

private volatile boolean flag = true;

**15. Lock Interface (Advanced Synchronization)**

Java provides the Lock interface (java.util.concurrent.locks.**Lock**) for more advanced thread synchronization than what synchronized can provide.

Locks provide flexibility in acquiring and releasing locks explicitly.

Lock lock = new ReentrantLock();

lock.lock();

try {

// Critical section

}

finally {

lock.unlock();

}

**16. Executor Framework**

The Executor framework provides a high-level mechanism for managing thread execution.

It separates thread management from the task itself.

Some key components are:

* **Executor**: A simple interface to execute tasks.
* **ExecutorService**: Provides lifecycle management of tasks and thread pools.
* **ScheduledExecutorService**: Executes tasks at scheduled intervals.

**Summary**

Threading concepts in Java provide powerful tools to execute tasks concurrently, manage shared resources, and control the flow of execution across multiple threads.

Proper understanding and usage of synchronization, thread communication, and concurrent utilities help in writing efficient and bug-free multithreaded programs.

**What do you mean by preemption?**

**Preemption** in the context of operating systems and threading refers to the ability of the operating system to interrupt and temporarily halt the execution of a running thread or process, in order to allocate CPU resources to another thread or process.

This interruption allows the operating system to manage the execution of multiple threads and processes efficiently, ensuring that critical tasks receive CPU time and that no single thread or process monopolizes the CPU.

**Key Aspects of Preemption:**

1. **Context Switching**:
   * When preemption occurs, the operating system saves the state (context) of the currently running thread or process, including its program counter, registers, and other execution details.
   * The operating system then restores the state of the next thread or process that is to be executed.
2. **Time Slicing**:
   * Preemption is often used in conjunction with a time-slicing mechanism, where each thread or process is given a small time slot (quantum) to execute.
   * After its time slice expires, the currently running thread is preempted to allow another thread to run, promoting fairness and responsiveness.
3. **Preemptive Multitasking**:
   * Modern operating systems use preemptive multitasking, which means the OS decides when a thread or process should be paused and another one should run, rather than relying on the threads or processes to yield control voluntarily.
   * This ensures that the system remains responsive and can handle multiple tasks simultaneously.
4. **Thread Priorities**:
   * Threads can have different priorities. The scheduler may use these priorities to decide which thread to run next.
   * A higher-priority thread can preempt a lower-priority thread if it becomes ready to run.

**Why Preemption is Important:**

* **Responsiveness**: It allows the system to be more responsive, especially in environments where user interaction is critical (e.g., GUIs).
* **Fairness**: Ensures that all threads and processes get a fair share of CPU time, preventing any single thread from hogging the CPU.
* **Concurrency**: Enables multiple threads to appear to run simultaneously, improving the efficiency of multi-core systems.

**Example Scenario:**

* In a preemptive multitasking system, if a thread is executing a long-running task but a higher-priority thread becomes ready (e.g., a system interrupt or an I/O operation completes), the operating system can preempt the currently running thread and switch to the higher-priority one. This ensures that important tasks are handled promptly.

**Preemption in Java:**

* In Java, the JVM and the underlying operating system manage thread scheduling. Java threads may be preempted if the JVM and OS support preemptive multitasking.
* The Thread.yield() method in Java is a hint to the thread scheduler that the current thread is willing to yield its current use of the CPU, though it's not a guarantee of preemption.

**Thread or Runnable?**

Choosing between extending the Thread class or implementing the Runnable interface to create a thread depends on the specific needs of your application.

However, in most cases, **implementing the Runnable interface** is considered the better practice.

Here's why:

**1. Separation of Concerns**

* **Runnable**: When you implement Runnable, you're simply defining the task that the thread will execute (run() method). This approach separates the task (the work to be done) from the threading mechanism.
* **Thread**: If you extend Thread, your class inherits all the thread-related methods. This ties the task directly to the thread, which can lead to less flexible and less reusable code.

**2. Java's Single Inheritance**

* **Runnable**: Java does not support multiple inheritance (a class can't extend more than one class). By implementing Runnable, you still have the flexibility to extend another class if needed.
* **Thread**: Extending Thread means you can't inherit from any other class, limiting the design possibilities.

**3. Better Resource Management**

* **Runnable**: Implementing Runnable allows you to use an ExecutorService to manage threads, which is more efficient and provides better resource management, especially for a large number of tasks.
* **Thread**: Directly managing threads through the Thread class can be more cumbersome and less scalable compared to using thread pools and executors.

**4. Reusability**

* **Runnable**: The same Runnable instance can be passed to multiple threads, allowing for more reusable and cleaner code.
* **Thread**: If you extend Thread, the task is tied to that particular thread instance, making reuse more difficult.

**Example Code**

**Using Runnable (Preferred Approach)**

class MyTask implements Runnable {

@Override

public void run() {

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

}

}

public class RunnableExample {

public static void main(String[] args) {

MyTask task = new MyTask();

Thread thread = new Thread(task);

thread.start();

}

}

**Using Thread (Less Preferred)**

class MyThread extends Thread {

@Override

public void run() {

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

}

}

public class ThreadExample {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start();

}

}

**Summary**

* **Use Runnable**: Preferred for most cases because it promotes a clearer separation between the task and the thread, allows for more flexible class design (since you can still extend another class), and is more compatible with thread pooling and executors.
* **Use Thread**: May be useful in simple, single-use cases where you do not need to extend another class and the task is closely tied to thread management. However, it's generally less flexible and not recommended for complex applications.

**Best Practice**

For modern Java applications, it is recommended to implement Runnable (or Callable for tasks that return a value) and use an ExecutorService to manage threads, which provides better control and resource management.

**How can you control or avoid deadlock?**

Deadlock occurs when two or more threads are blocked forever, each waiting on the other to release a resource.

To control or avoid deadlocks, several strategies can be employed:

**1. Avoid Nested Locks**

* Avoid acquiring multiple locks at the same time. The more locks you hold, the higher the risk of deadlock.
* If nested locks are necessary, make sure all threads acquire the locks in the same order.

**2. Lock Ordering**

* Establish a global ordering on the acquisition of locks and ensure all threads follow this order.
* If every thread acquires locks in a consistent order, circular waiting (a condition for deadlock) can be avoided.

**3. Lock Timeout**

* Use lock timeout mechanisms where a thread attempts to acquire a lock for a certain period. If it fails, it releases any locks it holds and retries.
* In Java, **ReentrantLock** provides a tryLock(long timeout, TimeUnit unit) method which can be used for this purpose:

ReentrantLock lock1 = new ReentrantLock();

ReentrantLock lock2 = new ReentrantLock();

if (lock1.tryLock(1000, TimeUnit.MILLISECONDS)) {

try {

if (lock2.tryLock(1000, TimeUnit.MILLISECONDS)) {

try {

// Critical section

} finally {

lock2.unlock();

}

}

} finally {

lock1.unlock();

}

}

**4. Use a Lock Hierarchy**

* Design a hierarchy for locks and acquire them in a predefined order.
* For example, always acquire lock A before lock B. This prevents circular waiting since threads will never wait for a lower-level lock while holding a higher-level one.

**5. Avoid Long-Running Locks**

* Keep the synchronized block or critical section as short as possible.
* Perform non-critical operations outside the locked section to minimize the time locks are held.

**6. Deadlock Detection and Recovery**

* Implement deadlock detection mechanisms that monitor the state of the system and detect cycles in the wait-for graph.
* If a deadlock is detected, take action such as terminating one or more threads to break the cycle.

**7. Lock-Free and Concurrent Data Structures**

* Use lock-free algorithms or concurrent data structures provided by the Java java.util.concurrent package.
* Classes like ConcurrentHashMap, ConcurrentLinkedQueue, and Atomic variables reduce the need for explicit synchronization, thus lowering the risk of deadlock.

**8. Avoid Resource Starvation**

* Make sure that threads have fair access to resources.
* Use fair locks (new ReentrantLock(true)) to ensure that locks are granted in the order they were requested.

**9. Consistent Locking Order in the Application**

* If you have a multi-threaded environment where threads need to acquire multiple locks, ensure that all threads acquire them in a consistent global order.

**Example of Lock Ordering**

Here’s an example demonstrating how to avoid deadlock using consistent lock ordering:

class Account {

private int balance = 1000;

public int getBalance() {

return balance;

}

public void deposit(int amount) {

balance += amount;

}

public void withdraw(int amount) {

balance -= amount;

}

public static void transfer(Account fromAccount, Account toAccount, int amount) {

synchronized (fromAccount) {

synchronized (toAccount) {

fromAccount.withdraw(amount);

toAccount.deposit(amount);

}

}

}

}

In this example:

* **Problem**: If two threads try to transfer money between two accounts at the same time, they might deadlock if they lock the accounts in a different order.
* **Solution**: Always lock the accounts in a consistent order, such as by using the account's ID to determine the lock order:

public static void transfer(Account fromAccount, Account toAccount, int amount) {

Account lock1 = fromAccount;

Account lock2 = toAccount;

if (fromAccount.hashCode() > toAccount.hashCode()) {

lock1 = toAccount;

lock2 = fromAccount;

}

synchronized (lock1) {

synchronized (lock2) {

fromAccount.withdraw(amount);

toAccount.deposit(amount);

}

}

}

**Summary**

* Deadlocks can be avoided by careful design of the locking mechanisms and following best practices such as lock ordering, timeouts, and minimizing the scope of synchronized blocks.
* Using higher-level concurrency utilities provided by the java.util.concurrent package can also help mitigate the risks of deadlocks.
* Implementing deadlock detection and recovery mechanisms can provide additional safety in complex multi-threaded applications.