Java Concurrency: Runnable vs Callable, ExecutorService, Synchronization, and Common Errors

---

1. Runnable vs Callable

- **Runnable**: No return value, cannot throw checked exceptions.
- **Callable**: Returns a value, can throw checked exceptions.

Example:

```
Runnable r = () -> System.out.println("Runnable");
Callable<Integer> c = () -> 42;
```

---

2. ExecutorService (Thread Pool)

- Manages a pool of threads for executing tasks.
- Use `submit()` for both Runnable and Callable.
- Always shut down the executor to free resources.

---

3. Future, Callable, and Exception Handling

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args)
            throws Exception {
        ExecutorService executor =
            Executors.newFixedThreadPool(1);

        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // System.out.println(future.get());
        // Compile error: must handle
        // InterruptedException, ExecutionException

        try {
            System.out.println(future.get());
            // Correct usage
        } catch (InterruptedException
```

```
                | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Common Error:

- Not handling checked exceptions from `future.get()`.
- Not shutting down the executor.

---

4. wait/notify and IllegalMonitorStateException

```
import java.util.*;

class Shared {
    synchronized void doWork()
            throws InterruptedException {
        System.out.println("Start");
        wait();
        System.out.println("End");
    }
}

public class Main {
    public static void main(String[] args) {
        Shared shared = new Shared();

        new Thread(() -> {
            try {
                shared.doWork();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        // shared.notify();
        // Error: IllegalMonitorStateException
        // if not in synchronized block
    }
}
```

Common Error:

- Calling `notify()` or `wait()` outside a synchronized block throws `IllegalMonitorStateException`.

---

5. notify vs notifyAll

```
import java.util.*;

class Shared {
    synchronized void doWork()
            throws InterruptedException {
        System.out.println(
            Thread.currentThread().getName()
            + " is doing work"
        );
        wait();
        System.out.println(
            Thread.currentThread().getName()
            + " has finished work"
        );
    }
}

public class Main {
    public static void main(String[] args)
            throws Exception {
        Shared shared = new Shared();

        Runnable r = () -> {
            try {
                shared.doWork();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        Thread t1 = new Thread(r, "T1");
        Thread t2 = new Thread(r, "T2");

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

        Thread.sleep(1000);

        synchronized(shared) {
```

```
            shared.notify();
            // Only one thread wakes up

            shared.notifyAll();
            // All waiting threads wake up
        }
    }
}
```

Common Error:

- Using `notify()` when multiple threads are waiting may leave some threads blocked.
- Always use `while` (not `if`) to check wait conditions to avoid spurious wakeups.

---

6. Exception Handling in ExecutorService

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        ExecutorService ex =
            Executors.newSingleThreadExecutor();

        Future<String> f = ex.submit(() -> {
            throw new IllegalStateException("Fail");
        });

        try {
            System.out.println(f.get());
        } catch (Exception e) {
            System.out.println(e.getClass());
            // ExecutionException

            System.out.println(e.getCause());
            // IllegalStateException: Fail
        } finally {
            ex.shutdown();
        }
    }
}
```

Explanation:

- Exceptions thrown in tasks are wrapped in `ExecutionException`.

- Use `getCause()` to get the original exception.

---

7. wait/notify with Condition Variable

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

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

    public synchronized void waitForValue(int target)
            throws InterruptedException {
        while(count < target) {
            // Use while, not if
            wait();
        }
        System.out.println("Reached: " + target);
    }
}

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

        new Thread(() -> {
            try {
                c.waitForValue(5);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }).start();

        for(int i = 0; i < 5; i++) {
            Thread.sleep(100);
            c.increment();
        }
    }
}
```

Common Error:

- Using `if` instead of `while` for wait condition can cause missed notifications or spurious wakeups.

---

8. Deadlock Example

```
public class Main {
    public static void main(String[] args) {
        final Object lock1 = new Object();
        final Object lock2 = new Object();

        Thread t1 = new Thread(() -> {
            synchronized(lock1) {
                try {
                    Thread.sleep(100);
                } catch (Exception e) {}
                synchronized(lock2) {
                    System.out.println(
                        "Thread 1 acquired both locks"
                    );
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized(lock2) {
                try {
                    Thread.sleep(100);
                } catch (Exception e) {}
                synchronized(lock1) {
                    System.out.println(
                        "Thread 2 acquired both locks"
                    );
                }
            }
        });

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

        // Deadlock: Both threads wait
        // for each other forever
    }
}
```

Common Error:

- Acquiring locks in different orders can cause deadlock.

---

9. Future, Runnable, Callable, and Exception Handling

```
public class Main {
    public static void main(String[] args)
            throws InterruptedException {
        ExecutorService ex =
            Executors.newFixedThreadPool(2);

        Runnable r1 = () -> {};
        Runnable r2 = () -> {
            throw new RuntimeException("boom");
        };

        Callable<String> c1 = () -> "ok";
        Callable<String> c2 = () -> {
            throw new IllegalStateException("bad");
        };

        Future<?> fr1 = ex.submit(r1);
        Future<?> fr2 = ex.submit(r2);
        Future<String> fc1 = ex.submit(c1);
        Future<String> fc2 = ex.submit(c2);

        try {
            System.out.println(fr1.get());
        } catch (ExecutionException e) {
            System.out.println("R1 : " + e);
        }

        try {
            System.out.println(fr2.get());
        } catch (ExecutionException e) {
            System.out.println("R2 : "
                + e.getClass());
        }

        try {
            System.out.println(fc1.get());
        } catch (ExecutionException e) {
            System.out.println("C1 : " + e);
        }
```

```
        try {
            System.out.println(fc2.get());
        } catch (ExecutionException e) {
            System.out.println("C2 : "
                + e.getCause());
        }

        ex.shutdown();
    }
}
```

Explanation:

- `ExecutionException` wraps exceptions thrown by tasks.
- `getCause()` reveals the original exception.

---

10. wait/notify and InterruptedException

```
public class Main {
    public static void main(String[] args)
            throws InterruptedException {
        Object lock = new Object();

        Thread t = new Thread(() -> {
            synchronized(lock) {
                try {
                    lock.wait();
                    System.out.println("Woke normally");
                } catch(InterruptedException e) {
                    System.out.println(
                        "Woke due to interruption"
                    );
                    System.out.println(
                        Thread.currentThread()
                              .isInterrupted()
                    );
                    // false
                }
            }
        });

        t.start();
        Thread.sleep(100);
        t.interrupt();
    }
```

```
}
```

Explanation:

- When a thread is interrupted during `wait()`, it throws `InterruptedException` and clears the interrupted status.
- `isInterrupted()` returns `false` in the catch block.

---

11. Error: notify on Wrong Object

```
class Box {
    private boolean ready = false;
    private final Object m1 = new Object();
    private final Object m2 = new Object();

    void awaitReady()
            throws InterruptedException {
        synchronized (m1) {
            if (!ready) {
                m1.wait();
            }
            System.out.println("go");
        }
    }

    void signalReady() {
        synchronized (m2) {
            ready = true;
            m2.notifyAll();
            // Wont work as m1 is waiting.
            // notifyAll is being called on m2
            // rather than m1
        }
    }
}

public class Main {
    public static void main(String[] args)
            throws InterruptedException {
        Box b = new Box();

        new Thread(() -> {
            try {
                b.awaitReady();
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }).start();

        Thread.sleep(1000);
        b.signalReady();
        // Thread will hang
    }
}
```

Common Error:

- Calling `notifyAll()` on a different object than the one used for `wait()` will not wake up the waiting thread.

Java Concurrency: Runnable vs Callable, ExecutorService, Synchronization, and Common Errors

---

1. Runnable vs Callable

- **Runnable**: No return value, cannot throw checked exceptions.
- **Callable**: Returns a value, can throw checked exceptions.

Example:

```
Runnable r = () -> System.out.println("Runnable");
Callable<Integer> c = () -> 42;
```

---

2. ExecutorService (Thread Pool)

- Manages a pool of threads for executing tasks.
- Use `submit()` for both Runnable and Callable.
- Always shut down the executor to free resources.

---

3. Future, Callable, and Exception Handling

```
import java.util.concurrent.*;
public class Main {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(1);
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });
        // System.out.println(future.get()); // Compile error: must handle InterruptedExcep
        try {
            System.out.println(future.get()); // Correct usage
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Common Error: - Not handling checked exceptions from `future.get()`. - Not shutting down the executor.

---

4. wait/notify and IllegalMonitorStateException

```
import java.util.*;
class Shared {
    synchronized void doWork() throws InterruptedException {
        System.out.println("Start");
        wait();
        System.out.println("End");
    }
}
public class Main {
    public static void main(String[] args) {
        Shared shared = new Shared();
        new Thread(() -> {
            try { shared.doWork(); } catch (InterruptedException e) { e.printStackTrace(); }
        }).start();
        // shared.notify(); // Error: IllegalMonitorStateException if not in synchronized b
    }
}
```

Common Error: - Calling `notify()` or `wait()` outside a synchronized block throws `IllegalMonitorStateException`.

---

5. notify vs notifyAll

```
import java.util.*;
class Shared {
    synchronized void doWork() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " is doing work");
        wait();
        System.out.println(Thread.currentThread().getName() + " has finished work");
    }
}
public class Main {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Runnable r = () -> {
            try { shared.doWork(); } catch (InterruptedException e) { e.printStackTrace(); }
        };
        Thread t1 = new Thread(r, "T1");
        Thread t2 = new Thread(r, "T2");
        t1.start(); t2.start();
        Thread.sleep(1000);
```

```
        synchronized(shared) {
            shared.notify();    // Only one thread wakes up
            shared.notifyAll(); // All waiting threads wake up
        }
    }
}
```

Common Error: - Using `notify()` when multiple threads are waiting may leave some threads blocked. - Always use `while` (not `if`) to check wait conditions to avoid spurious wakeups.

---

6. Exception Handling in ExecutorService

```
import java.util.concurrent.*;
public class Main {
    public static void main(String[] args) {
        ExecutorService ex = Executors.newSingleThreadExecutor();
        Future<String> f = ex.submit(() -> { throw new IllegalStateException("Fail"); });
        try {
            System.out.println(f.get());
        } catch (Exception e) {
            System.out.println(e.getClass());   // ExecutionException
            System.out.println(e.getCause());   // IllegalStateException: Fail
        } finally {
            ex.shutdown();
        }
    }
}
```

Explanation: - Exceptions thrown in tasks are wrapped in `ExecutionException`. - Use `getCause()` to get the original exception.

---

7. wait/notify with Condition Variable

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
        notifyAll();
    }
    public synchronized void waitForValue(int target) throws InterruptedException {
        while(count < target) { // Use while, not if
            wait();
        }
```

```
        System.out.println("Reached: " + target);
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        new Thread(() -> {
            try { c.waitForValue(5); } catch(Exception e) { e.printStackTrace(); }
        }).start();
        for(int i = 0; i < 5; i++) {
            Thread.sleep(100);
            c.increment();
        }
    }
}
```

Common Error: - Using `if` instead of `while` for wait condition can cause missed notifications or spurious wakeups.

---

8. Deadlock Example

```
public class Main {
    public static void main(String[] args) {
        final Object lock1 = new Object();
        final Object lock2 = new Object();
        Thread t1 = new Thread(() -> {
            synchronized(lock1) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized(lock2) { System.out.println("Thread 1 acquired both locks"); }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized(lock2) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized(lock1) { System.out.println("Thread 2 acquired both locks"); }
            }
        });
        t1.start(); t2.start();
        // Deadlock: Both threads wait for each other forever
    }
}
```

Common Error: - Acquiring locks in different orders can cause deadlock.

---

9. Future, Runnable, Callable, and Exception Handling

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService ex = Executors.newFixedThreadPool(2);
        Runnable r1 = () -> {};
        Runnable r2 = () -> { throw new RuntimeException("boom"); };
        Callable<String> c1 = () -> "ok";
        Callable<String> c2 = () -> { throw new IllegalStateException("bad"); };
        Future<?> fr1 = ex.submit(r1);
        Future<?> fr2 = ex.submit(r2);
        Future<String> fc1 = ex.submit(c1);
        Future<String> fc2 = ex.submit(c2);
        try { System.out.println(fr1.get()); } catch (ExecutionException e) { System.out.pri
        try { System.out.println(fr2.get()); } catch (ExecutionException e) { System.out.pri
        try { System.out.println(fc1.get()); } catch (ExecutionException e) { System.out.pri
        try { System.out.println(fc2.get()); } catch (ExecutionException e) { System.out.pri
        ex.shutdown();
    }
}
```

Explanation: - `ExecutionException` wraps exceptions thrown by tasks. - `getCause()` reveals the original exception.

---

10. wait/notify and InterruptedException

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();
        Thread t = new Thread(() -> {
            synchronized(lock) {
                try {
                    lock.wait();
                    System.out.println("Woke normally");
                } catch(InterruptedException e) {
                    System.out.println("Woke due to interruption");
                    System.out.println(Thread.currentThread().isInterrupted()); // false
                }
            }
        });
        t.start();
        Thread.sleep(100);
        t.interrupt();
    }
}
```

Explanation: - When a thread is interrupted during `wait()`, it throws `InterruptedException` and clears the interrupted status. - `isInterrupted()` returns `false` in the catch block.

---

11. Error: notify on Wrong Object

```
class Box {
    private boolean ready = false;
    private final Object m1 = new Object();
    private final Object m2 = new Object();
    void awaitReady() throws InterruptedException {
        synchronized (m1) {
            if (!ready) {
                m1.wait();
            }
            System.out.println("go");
        }
    }
    void signalReady() {
        synchronized (m2) {
            ready = true;
            m2.notifyAll(); // Wont work as m1 is waiting. notifyAll is being called on m2 t
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Box b = new Box();
        new Thread(() -> {
            try { b.awaitReady(); } catch (InterruptedException e) { e.printStackTrace(); }
        }).start();
        Thread.sleep(1000);
        b.signalReady(); // Thread will hang
    }
}
```

Common Error: - Calling `notifyAll()` on a different object than the one used for `wait()` will not wake up the waiting thread.
