

JAVA PROGRAMMING

Chapter 4

Multithreaded Programming and Exception Handling

4.1 Multi Threading

4.2 Managing Errors and Exceptions

Detailed Notes for Engineering Students

Theory + Programs + Tables + 15 MCQs with Answer Key + 10 Broad Questions

PolyNotesHub

Contents

4.1 Multi Threading	3
4.1.1 Introduction	3
4.1.2 Creating Threads	3
4.1.3 Extending the Thread Class	4
4.1.4 Implementing the Runnable Interface	5
4.1.5 Stopping and Blocking a Thread	6
4.1.6 Life Cycle of a Thread	7
4.1.7 Using Thread Methods	9
4.1.8 Thread Exceptions	9
4.1.9 Thread Priority	10
4.1.10 Synchronization	11
4.2 Managing Errors and Exceptions	14
4.2.1 Types of Errors	14
4.2.2 Exception	14
4.2.3 Multiple Catch Statements	17
4.2.4 Using the finally Statement	19
4.2.5 Using Exceptions for Debugging	20
Multiple Choice Questions (15)	23
Answer Key	25
Broad Questions (10)	26

4.1 Multi Threading

4.1.1 Introduction

Multitasking means performing more than one job at the same time. A modern computer plays music, downloads a file and runs a word processor together. Multitasking can be of two kinds: **process-based** (several programs run together) and **thread-based** (several parts of the same program run together).

A **thread** is a lightweight, independent path of execution inside a program. A program that contains two or more threads running concurrently is called a **multithreaded program**. Every Java program has at least one thread, the **main thread**, which is created automatically by the JVM and executes the `main()` method. Other threads are created by the programmer.

Threads of one program share the same memory (heap), so communication between them is cheap, and switching between threads is faster than switching between processes. Web servers, games, GUI programs and animation make heavy use of threads. On a machine with a single CPU the threads are executed one after another in very short slices, which looks like parallel execution.

Process and thread

Basis	Process	Thread
Meaning	A program in execution	A part (path of execution) of a process
Memory	Has its own separate memory space	Shares the memory of the parent process
Weight	Heavyweight	Lightweight
Communication	Slow, needs inter-process mechanisms	Fast, through shared objects
Context switch	Costly	Cheap
Creation cost	High	Low

Table 4.1: Process vs Thread

Advantages of multithreading

- Better utilisation of the CPU: while one thread waits for input, another can run.
- Programs stay **responsive**; a long task does not freeze the user interface.
- Threads share resources, so they use less memory than separate processes.
- Simple modelling of real-world parallel activities (animation, downloads, servers).
- Faster execution on multi-core processors.

4.1.2 Creating Threads

Java supports threads through the class `java.lang.Thread` and the interface `java.lang.Runnable`. Both contain the method `public void run()`, which holds the code that the thread executes. There are two ways to create a thread:

Way	Steps
1. Extending the Thread class	Create a class that extends Thread, override run(), create an object of the class and call start().
2. Implementing the Runnable interface	Create a class that implements Runnable, define run(), pass its object to a Thread constructor and call start().

Table 4.2: Two ways of creating a thread

4.1.3 Extending the Thread Class

In this method the new class is a subclass of Thread. The general form is:

Syntax

```
class MyThread extends Thread
{
    public void run()
    {
        // code to be executed by the thread
    }
}
```

Steps

1. Declare the class as a subclass of Thread.
2. Override the run() method with the code the thread should perform.
3. Create an object of the class: `MyThread t = new MyThread();` (the thread is now in the **newborn** state).
4. Call `t.start()`. The JVM creates a new call stack and then calls run() on that thread.

Example 1: Creating a thread by extending Thread

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 3; i++)
            System.out.println("Child thread: " + i);
    }
}
public class T1
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t = new MyThread();
        t.start();
        t.join();           // main waits for the child thread
        System.out.println("Main thread finished");
    }
}
```

Output

```
Child thread: 1
Child thread: 2
Child thread: 3
Main thread finished
```

start() versus run()

Point	start()	run()
Effect	Creates a new thread and then calls run() in it	Runs like an ordinary method in the current thread
Threads used	New thread is created	No new thread is created
Calling twice	IllegalThreadStateException	Allowed (just a method call)
Purpose	Begin concurrent execution	Holds the work of the thread

Table 4.3: start() vs run()

Note: Never call run() directly if you want multithreading. Always call start().

4.1.4 Implementing the Runnable Interface

Java does not allow a class to extend more than one class. If a class already extends another class (for example an applet or a frame), it cannot also extend Thread. In such a case the class implements the Runnable interface, which has a single method run(). The Runnable object is then given to a Thread object.

Example 2: Creating a thread by implementing Runnable

```
class Task implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        for (int i = 1; i <= 3; i++)
            System.out.println(t.getName() + " : " + i);
    }
}
public class T2
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread w = new Thread(new Task(), "Worker-1");
        w.start();
        w.join();
        System.out.println("Done");
    }
}
```

Output

```
Worker-1 : 1
Worker-1 : 2
Worker-1 : 3
Done
```

Thread class vs Runnable interface

Basis	Extending Thread	Implementing Runnable
Inheritance	Class cannot extend any other class	Class is free to extend another class
Type	Thread is a class	Runnable is an interface
Object sharing	Each thread object is a separate object	One Runnable object can be shared by many threads
Method to define	run()	run()
Start	obj.start()	new Thread(obj).start()
Preferred when	Simple programs	Class already has a parent; better design (separates task from thread)

Table 4.4: Thread vs Runnable

4.1.5 Stopping and Blocking a Thread

Stopping a thread

A thread stops (dies) automatically when its `run()` method finishes. Sometimes it has to be stopped earlier. The old method `stop()` is **deprecated** because it ends the thread abruptly and can leave shared data in an inconsistent state. The recommended ways are:

- Use a **volatile boolean flag** that the `run()` loop checks regularly. Another thread sets it to false.
- Call `interrupt()` on the thread. A sleeping or waiting thread receives an **InterruptedException** and can finish cleanly.

Blocking a thread

A thread is **blocked** when it is temporarily stopped from running but is not dead. It can return to the runnable state afterwards. A thread can be blocked in the following ways:

Method / situation	Effect	How it comes back
<code>sleep(ms)</code>	Thread sleeps for the given time	Automatically after the time elapses
<code>wait()</code>	Thread waits until another thread signals	<code>notify()</code> or <code>notifyAll()</code> is called
<code>join()</code>	Thread waits for another thread to finish	The other thread dies
Waiting for a lock	Thread tries to enter a synchronized block held by another	The lock is released
<code>suspend()</code> (deprecated)	Thread suspended indefinitely	<code>resume()</code> (deprecated, deadlock-prone)

Table 4.5: Ways of blocking a thread

Example 3: Stopping a thread with a flag and with interrupt()

```
class Worker implements Runnable
{
    private volatile boolean running = true;
    void stopWork() { running = false; }
    public void run()
    {
        while (running)
        {
            try { Thread.sleep(50); } catch (InterruptedException e) { }
        }
        System.out.println("Worker stopped");
    }
}
class Sleeper extends Thread
{
    public void run()
    {
        try
        {
            Thread.sleep(5000);
            System.out.println("Woke up normally");
        }
        catch (InterruptedException e)
        {
            System.out.println("Sleep interrupted");
        }
    }
}
public class T5
{
    public static void main(String[] args) throws InterruptedException
    {
        Worker w = new Worker();
        Thread t = new Thread(w);
        t.start();
        Thread.sleep(200);
        w.stopWork();           // stop using the flag
        t.join();
        Sleeper s = new Sleeper();
        s.start();
        s.interrupt();          // stop a sleeping thread
        s.join();
    }
}
```

Output

```
Worker stopped
Sleep interrupted
```

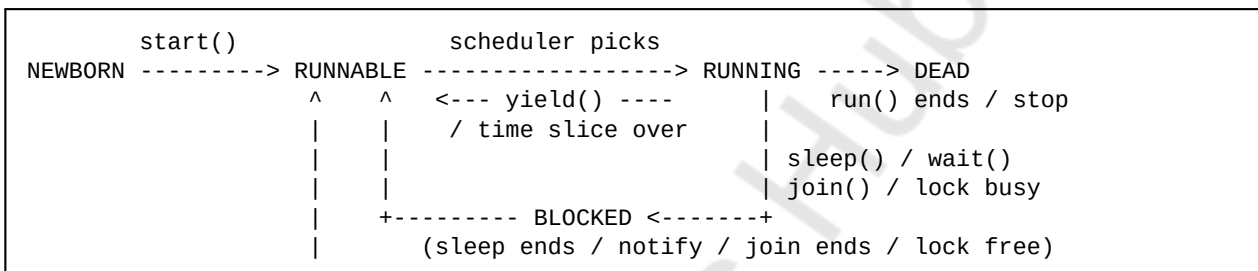
4.1.6 Life Cycle of a Thread

During its life a thread passes through several states. It can be in exactly one state at a time. The five states of the classic model are:

State	Meaning	How the thread enters it
Newborn (New)	Thread object created but not started	new Thread(...)
Runnable	Ready and waiting for the CPU in the queue	start(), or return from blocked state, or yield()
Running	Thread is executing its run() method	Chosen by the thread scheduler
Blocked (Waiting)	Temporarily inactive	sleep(), wait(), join(), waiting for a lock
Dead (Terminated)	Thread has finished	run() completes, or an uncaught exception ends it

Table 4.6: States of a thread

Figure 4.1: Life cycle of a thread



In the Java API, `Thread.getState()` returns one of the values of the enum `Thread.State`:

Thread.State value	Equivalent in the classic model
NEW	Newborn
RUNNABLE	Runnable and Running (Java does not separate them)
BLOCKED	Waiting to acquire a monitor lock
WAITING	Blocked in wait() or join() without a time limit
TIMED_WAITING	Blocked in sleep(ms), or wait/join with a time limit
TERMINATED	Dead

Table 4.7: Thread.State values

Example 4: Observing thread states

```

public class T3
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread t = new Thread(() -> {
            try { Thread.sleep(500); } catch (InterruptedException e) { }
        });
        System.out.println("Before start : " + t.getState());
        t.start();
        Thread.sleep(100);
        System.out.println("While sleeping: " + t.getState());
        t.join();
        System.out.println("After finish : " + t.getState());
        System.out.println("isAlive      : " + t.isAlive());
    }
}
  
```

Output

```
Before start : NEW
While sleeping: TIMED_WAITING
After finish : TERMINATED
isAlive      : false
```

4.1.7 Using Thread Methods

The Thread class provides many methods to control threads. The most important ones are listed below.

Method	Purpose
void start()	Starts the thread; JVM calls run()
void run()	Entry point; contains the work of the thread
static void sleep(long ms)	Pauses the current thread for the given milliseconds; throws InterruptedException
final void join()	Makes the calling thread wait until this thread dies
static void yield()	Hints that the current thread is willing to give up the CPU to others
final boolean isAlive()	True if the thread has started and not yet died
final String getName() / setName(String)	Gets or sets the thread's name
final int getPriority() / setPriority(int)	Gets or sets the priority (1 to 10)
static Thread currentThread()	Returns the thread that is executing the call
void interrupt()	Interrupts a thread; sleeping or waiting threads get InterruptedException
boolean isInterrupted()	Tests whether the thread has been interrupted
final void setDaemon(boolean)	Marks the thread as daemon (background) thread; must be called before start()
wait(), notify(), notifyAll()	Methods of class Object used for inter-thread communication; must be called inside synchronized code

- **Daemon threads** are service threads (for example the garbage collector). The JVM exits when only daemon threads remain.
- **join()** is used to make sure that one thread finishes before another continues, as in Example 1.
- **sleep()** does not release the lock held by the thread, whereas **wait()** releases it.

4.1.8 Thread Exceptions

Thread operations may throw exceptions. Some are **checked** exceptions and must be handled with try-catch or declared with throws; others are **unchecked** (run-time) exceptions.

Exception	Type	Cause
InterruptedException	Checked	A thread in sleep(), wait() or join() is interrupted by another thread
IllegalThreadStateException	Unchecked	Calling start() twice on the same thread, or setDaemon() after start
IllegalArgumentException	Unchecked	Priority outside the range 1 to 10 passed to setPriority()
IllegalMonitorStateException	Unchecked	wait(), notify() or notifyAll() called without owning the object's lock
SecurityException	Unchecked	The security manager does not permit the thread operation

Table 4.9: Common thread-related exceptions

Example 5: Thread exceptions

```

public class T7
{
    public static void main(String[] args)
    {
        Thread t = new Thread(() -> System.out.println("Running"));
        t.start();
        try { t.join(); } catch (InterruptedException e) { }
        try
        {
            t.start();                // second start
        }
        catch (IllegalThreadStateException e)
        {
            System.out.println("Caught: IllegalThreadStateException");
        }
        try
        {
            new Object().wait();      // without synchronized
        }
        catch (IllegalMonitorStateException e)
        {
            System.out.println("Caught: IllegalMonitorStateException");
        }
        catch (InterruptedException e) { }
    }
}

```

Output

```

Running
Caught: IllegalThreadStateException
Caught: IllegalMonitorStateException

```

4.1.9 Thread Priority

Each thread has a **priority**, an integer from 1 to 10, which tells the thread scheduler how important the thread is. In general, a thread with higher priority gets the CPU earlier than one with lower priority. A new thread **inherits** the priority of the thread that created it. The default priority is 5 (that of the main thread).

Constant	Value	Meaning
Thread.MIN_PRIORITY	1	Lowest priority
Thread.NORM_PRIORITY	5	Default priority
Thread.MAX_PRIORITY	10	Highest priority

Table 4.10: Priority constants

- Priority is set with `setPriority(int)` and read with `getPriority()`.
- A value outside 1 to 10 throws **IllegalArgumentException**.
- Priority is only a **hint** to the scheduler. Behaviour depends on the operating system, so a program must never depend on priority for correctness.
- Java uses preemptive scheduling (a higher-priority runnable thread can take the CPU) and, on most systems, time-slicing among threads of equal priority.

Example 6: Thread priority

```
public class T4
{
    public static void main(String[] args)
    {
        Thread m = Thread.currentThread();
        System.out.println("Name      : " + m.getName());
        System.out.println("Priority : " + m.getPriority());
        Thread t = new Thread();
        t.setPriority(Thread.MAX_PRIORITY);
        System.out.println("MIN=" + Thread.MIN_PRIORITY + " NORM=" +
                           Thread.NORM_PRIORITY + " MAX=" + Thread.MAX_PRIORITY);
        System.out.println("t priority: " + t.getPriority());
        try { t.setPriority(11); }
        catch (IllegalArgumentException e)
        { System.out.println("Error: IllegalArgumentException"); }
    }
}
```

Output

```
Name      : main
Priority : 5
MIN=1 NORM=5 MAX=10
t priority: 10
Error: IllegalArgumentException
```

4.1.10 Synchronization

When two or more threads use the same data (a shared resource) at the same time and at least one of them changes it, the result can be wrong. This situation is called a **race condition**. For example, `count++` is really three steps (read, add, write); if two threads interleave these steps, one update is lost.

Synchronization ensures that only **one thread at a time** can use a shared resource. Every Java object has an internal **lock** (monitor). A thread must obtain the lock before entering a synchronized method or block, and releases it on leaving. Other threads that need the same lock are blocked until it is free.

Ways to synchronize

Syntax

```
// 1. synchronized method
synchronized void increment() { count++; }
// 2. synchronized block
void increment()
{
    synchronized (this)
    {
        count++;
    }
}
```

Basis	Synchronized method	Synchronized block
Scope	Whole method	Only the selected statements
Lock used	this (or the Class object for static methods)	Any object named in the parentheses
Performance	Locks for the entire method	Better; smaller critical section
Use when	Whole method touches shared data	Only a few statements touch shared data

Table 4.11: Synchronized method vs block

Example 7: Two threads updating one counter safely

```
class Counter
{
    private int count = 0;
    synchronized void increment() { count++; }
    int getCount() { return count; }
}
class Inc extends Thread
{
    Counter c;
    Inc(Counter c) { this.c = c; }
    public void run()
    {
        for (int i = 0; i < 1000; i++)
            c.increment();
    }
}
public class T6
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter c = new Counter();
        Inc t1 = new Inc(c);
        Inc t2 = new Inc(c);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println("Final count = " + c.getCount());
    }
}
```

Output

```
Final count = 2000
```

If the keyword `synchronized` is removed from `increment()`, the final count will often be less than 2000 because updates are lost.

Deadlock

A **deadlock** occurs when two threads each hold a lock and wait for the lock held by the other, so neither can proceed. It can be avoided by acquiring locks in the same order in all threads and by keeping synchronized sections short.

Points to remember: Multithreading

- Create a thread by extending Thread or implementing Runnable; always start it with start().
- run() holds the thread's work; calling run() directly does not create a thread.
- A thread cannot be started twice; doing so throws IllegalStateException.
- Priority ranges from 1 to 10; default is 5; it is only a hint to the scheduler.
- sleep() and join() throw the checked exception InterruptedException.
- Do not use stop(), suspend() and resume(); they are deprecated. Use a flag or interrupt().
- Use synchronized to protect shared data from race conditions.

4.2 Managing Errors and Exceptions

4.2.1 Types of Errors

An **error** is a mistake in a program that makes it behave in an unwanted way or prevents it from running. Errors are mainly of two types: **compile-time errors** and **run-time errors**. A third kind, the logical error, is also discussed in practice.

Type	When it occurs	Detected by	Examples
Compile-time (syntax) error	While compiling the source code	Compiler (javac); no .class file is produced	Missing semicolon, undeclared variable, wrong data type, misspelt keyword, missing return
Run-time error	While the program is running	JVM; the program stops abnormally unless handled	Division by zero, array index out of range, invalid number format, file not found, null reference
Logical error	Program runs but gives wrong result	Programmer, by testing and debugging	Using + instead of *, wrong loop condition, wrong formula

Table 4.12: Types of errors

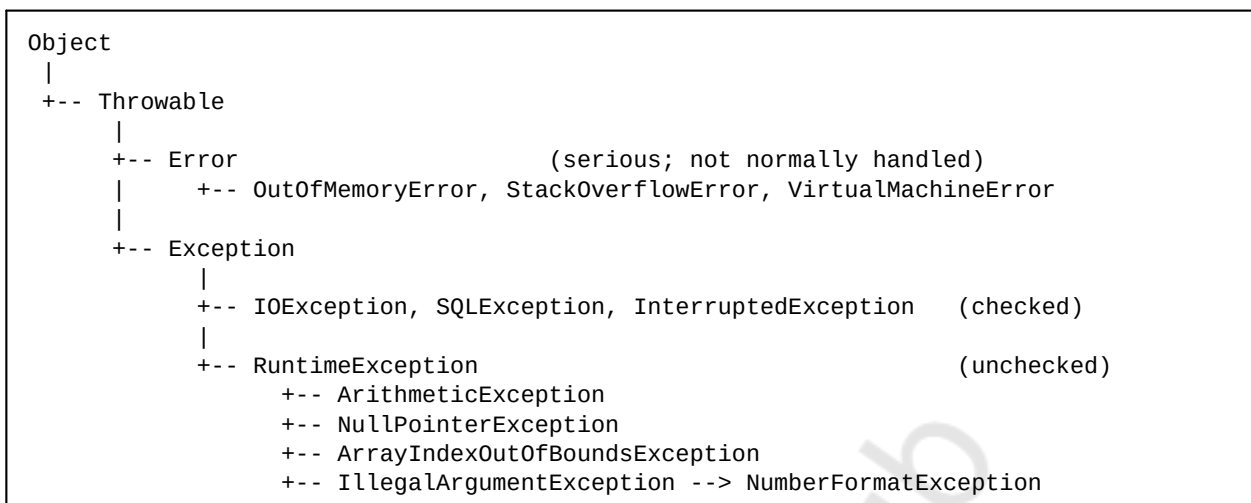
Common run-time problems

- Dividing an integer by zero.
- Accessing an element beyond the size of an array.
- Trying to store a value of an incompatible type in an array.
- Passing a parameter that is not in a valid range.
- Opening a file that does not exist, or a failing network connection.
- Converting an invalid string to a number.
- Using an object reference that is `null`.

4.2.2 Exception

An **exception** is an abnormal condition that arises at run time and disturbs the normal flow of the program. When an error occurs inside a method, Java creates an **exception object** containing information about the error (type, message, place) and **throws** it. If the program does not **catch** the exception, the default handler prints a message and the program terminates. **Exception handling** is the mechanism that lets the programmer detect the problem, report it and continue with the program in a controlled way.

Figure 4.2: Exception class hierarchy



Checked and unchecked exceptions

Basis	Checked exceptions	Unchecked exceptions
Checked at	Compile time	Run time
Class	Subclasses of Exception (except RuntimeException)	RuntimeException, Error and their subclasses
Must handle?	Yes: try-catch or declare with throws	No (optional)
Examples	IOException, SQLException, FileNotFoundException, InterruptedException	ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException

Table 4.13: Checked vs unchecked exceptions

Some common exceptions

Exception	Cause
ArithmeticException	Arithmetic error such as integer division by zero
ArrayIndexOutOfBoundsException	Array index negative or not less than the array length
NullPointerException	Using an object reference that is null
NumberFormatException	Converting a string that is not a valid number (Integer.parseInt("abc"))
ClassCastException	Invalid cast of an object to a class it is not an instance of
FileNotFoundException	Trying to open a file that does not exist
StringIndexOutOfBoundsException	Illegal index used with a String method
NegativeArraySizeException	Creating an array with a negative size

Using try and catch

The statements that may cause an exception are placed in a **try** block. The code to handle the exception is placed in a **catch** block immediately after it. When an exception occurs in the try block, the remaining statements of the try block are skipped and control jumps to the matching catch block.

Syntax

```
try
{
    // statements that may throw an exception
}
catch (ExceptionType e)
{
    // statements that handle the exception
}
```

Example 8: Handling ArithmeticException

```
public class E1
{
    public static void main(String[] args)
    {
        int a = 10, b = 0;
        try
        {
            int c = a / b;
            System.out.println(c);
        }
        catch (ArithmeticException e)
        {
            System.out.println("Error: " + e.getMessage());
            System.out.println(e);
        }
        System.out.println("Program continues");
    }
}
```

Output

```
Error: / by zero
java.lang.ArithmeticException: / by zero
Program continues
```

Keywords used in exception handling

Keyword	Use
try	Encloses the code that may throw an exception
catch	Handles a particular type of exception thrown in the try block
finally	Block that executes whether or not an exception occurs (cleanup)
throw	Explicitly throws an exception object: <code>throw new MyException("msg");</code>
throws	Declared in a method header to list checked exceptions the method may pass to its caller

Table 4.15: Exception-handling keywords

throw, throws and user-defined exceptions

A programmer can create his or her own exception class by extending `Exception` and throw it with the `throw` statement. A method that may throw a checked exception must declare it with `throws`.

Example 9: User-defined exception

```
class MyException extends Exception
{
    MyException(String msg) { super(msg); }
}
public class E4
{
    static void check(int age) throws MyException
    {
        if (age < 18)
            throw new MyException("Age " + age + " is below 18");
        System.out.println("Age " + age + " is valid");
    }
    public static void main(String[] args)
    {
        try
        {
            check(25);
            check(15);
        }
        catch (MyException e)
        {
            System.out.println("Caught: " + e.getMessage());
        }
        finally
        {
            System.out.println("End");
        }
    }
}
```

Output

```
Age 25 is valid
Caught: Age 15 is below 18
End
```

Basis	throw	throws
Purpose	Actually throws an exception object	Declares that a method may throw exceptions
Position	Inside the method body	In the method header
Followed by	An exception object (instance)	One or more exception class names
Count	Throws one exception at a time	Can list many exceptions separated by commas

Table 4.16: throw vs throws

4.2.3 Multiple Catch Statements

A single piece of code can cause more than one type of exception. To handle each type differently, several catch blocks can be written after one try block. When an exception occurs, the catch blocks are examined **in order**, and the **first** one whose type matches is executed; the others are skipped. Control then goes to the statement after the last catch block.

Syntax

```
try
{
    // statements
}
catch (ExceptionType1 e1) { // handler 1 }
catch (ExceptionType2 e2) { // handler 2 }
catch (Exception e)      { // general handler }
```

Example 10: Multiple catch blocks

```
public class E2
{
    public static void main(String[] args)
    {
        String[] data = {"10", "abc", "30"};
        for (int i = 0; i <= 3; i++)
        {
            try
            {
                int n = Integer.parseInt(data[i]);
                System.out.println("Value = " + n);
            }
            catch (NumberFormatException e)
            {
                System.out.println("Not a number: " + data[i]);
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Index error: " + e.getMessage());
            }
            catch (Exception e)
            {
                System.out.println("Other error");
            }
        }
    }
}
```

Output (the wording of the index message may differ slightly between Java versions)

```
Value = 10
Not a number: abc
Value = 30
Index error: Index 3 out of bounds for length 3
```

Rules

- Catch blocks for **subclass exceptions must come before** those for superclass exceptions. If a general catch such as `catch (Exception e)` is placed first, a later catch for a subclass is unreachable and the compiler reports an error (*exception has already been caught*).
- Only one catch block is executed for one exception.
- A try block must be followed by at least one catch block or a finally block.
- Since Java 7, one catch can handle several unrelated types using the pipe symbol: `catch (NumberFormatException | ArithmeticException e)`.
- Try blocks can be **nested**; if the inner block has no matching catch, the exception passes to the outer block.

4.2.4 Using the finally Statement

The **finally** block contains code that must be executed **whether or not an exception occurs**, and whether or not it is caught. It is used for clean-up work such as closing files, closing database connections or releasing resources. It is written after the last catch block (or directly after try).

Syntax

```
try
{
    // statements
}
catch (ExceptionType e)
{
    // handler
}
finally
{
    // clean-up code (always executed)
}
```

Example 11: try-catch-finally

```
public class E3
{
    static int test()
    {
        try
        {
            return 1;
        }
        finally
        {
            System.out.println("finally runs even after return");
        }
    }
    public static void main(String[] args)
    {
        try
        {
            int[] a = new int[2];
            a[5] = 1;
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Caught");
        }
        finally
        {
            System.out.println("Cleanup in finally");
        }
        System.out.println("test() returned " + test());
    }
}
```

Output

```
Caught
Cleanup in finally
finally runs even after return
test() returned 1
```

Situation	Does finally run?
No exception occurs	Yes
Exception occurs and is caught	Yes
Exception occurs and is not caught	Yes (then the exception propagates)
return statement in try or catch	Yes (finally runs before the method returns)
System.exit() is called in try	No (the JVM shuts down)
JVM crashes or power failure	No

Table 4.17: Execution of finally

- A try statement needs at least one of catch or finally; try-finally without catch is allowed.
- Only one finally block is allowed per try block.
- From Java 7, **try-with-resources** try (FileReader f = new FileReader("a.txt")) { ... } closes the resource automatically.

4.2.5 Using Exceptions for Debugging

Debugging is the process of finding and removing the cause of errors. Exceptions are a powerful debugging aid because the exception object records *what* went wrong and *where*. Information in a Throwable object can be displayed with the following methods:

Method	Returns / Action
String getMessage()	Short description of the problem
String toString()	Class name of the exception followed by the message
void printStackTrace()	Prints the exception and the chain of method calls (with line numbers) that led to it, on the error stream
StackTraceElement[] getStackTrace()	The stack trace as an array; each element gives class, method, file and line number
Throwable getCause()	The original exception that caused this one (if any)

Table 4.18: Methods of Throwable useful for debugging

Example 12: Reading the stack trace

```
public class E5
{
    static void level2() { int x = 5 / 0; }
    static void level1() { level2(); }
    public static void main(String[] args)
    {
        try { level1(); }
        catch (ArithmeticException e)
        {
            StackTraceElement[] st = e.getStackTrace();
            System.out.println("Message : " + e.getMessage());
            System.out.println("Thrown in: " + st[0].getMethodName());
            System.out.println("Called by: " + st[1].getMethodName());
            System.out.println("Called by: " + st[2].getMethodName());
        }
    }
}
```

Output

```
Message : / by zero
Thrown in: level2
Called by: level1
Called by: main
```

Debugging tips

- Read the **first line** of a stack trace for the exception type and message, and the **first line below it** for the exact place (file and line number).
- Use `e.printStackTrace()` inside catch blocks while developing so that errors are not silently hidden.
- Never write an empty catch block; at least print or log the exception.
- Print the values of important variables before the statement that fails.
- Use meaningful messages when throwing exceptions: `throw new IllegalArgumentException("age must be positive")`.
- In real applications, use a logging framework instead of `System.out.println`.

Points to remember: Exceptions

- An exception is an object thrown at run time when an abnormal condition occurs.
- Throwable is the root class; Error and Exception are its two subclasses.
- Checked exceptions must be handled or declared; unchecked exceptions need not be.
- In multiple catch blocks, write the specific (subclass) exception first and the general one last.
- `finally` always executes (except `System.exit` or a JVM crash) and is used for clean-up.
- `throw` throws an exception; `throws` declares exceptions in a method header.
- The stack trace is the most useful debugging aid provided by exceptions.

Common errors

Mistake	Result
catch (Exception e) before catch (ArithmeticException e)	Compile-time error: exception already caught
try block with neither catch nor finally	Compile-time error
Not handling a checked exception (e.g. Thread.sleep() without try or throws)	Compile-time error: unreported exception
Calling start() twice on a thread	IllegalThreadStateException at run time
Empty catch block	Error is hidden; debugging becomes very difficult

Table 4.19: Common mistakes

Multiple Choice Questions (15)

Choose the correct option. The answer key is given at the end of this section.

Q1. Which method is used to start a thread so that the run() method executes in a new thread?

- (A) run()
- (B) start()
- (C) begin()
- (D) execute()

Q2. Which interface must be implemented to create a thread when the class already extends another class?

- (A) Thread
- (B) Runnable
- (C) Serializable
- (D) Cloneable

Q3. If run() is called directly on a Thread object, then:

- (A) A new thread is created
- (B) The JVM throws an exception
- (C) run() executes like a normal method in the current thread
- (D) The thread goes to the dead state

Q4. What is the default priority of a thread in Java?

- (A) 0
- (B) 1
- (C) 5
- (D) 10

Q5. The valid range of thread priority is:

- (A) 0 to 9
- (B) 1 to 10
- (C) 1 to 5
- (D) 0 to 100

Q6. Which method makes the calling thread wait until another thread finishes?

- (A) sleep()
- (B) yield()
- (C) join()
- (D) interrupt()

Q7. Starting the same thread object twice throws:

- (A) IllegalArgumentException
- (B) InterruptedException
- (C) IllegalThreadStateException
- (D) NullPointerException

Q8. Which keyword is used to allow only one thread at a time to access a method?

- (A) static
- (B) final
- (C) volatile
- (D) synchronized

Q9. Which is the state of a thread that has completed its run() method?

- (A) Newborn
- (B) Runnable
- (C) Blocked
- (D) Dead

Q10. Which class is the direct superclass of both Error and Exception?

- (A) Object
- (B) Throwable
- (C) RuntimeException
- (D) Number

Q11. Which of the following is a checked exception?

- (A) ArithmeticException
- (B) NullPointerException
- (C) IOException
- (D) ArrayIndexOutOfBoundsException

Q12. In multiple catch blocks, the catch for a subclass exception must be placed:

- (A) After the superclass catch
- (B) Before the superclass catch
- (C) Anywhere
- (D) Inside the finally block

Q13. Which block is executed whether or not an exception occurs?

- (A) try
- (B) catch
- (C) finally
- (D) throw

Q14. Which keyword is used to explicitly throw an exception object?

- (A) throws
- (B) throw
- (C) catch
- (D) final

Q15. What exception occurs when an integer is divided by zero at run time?

- (A) NumberFormatException
- (B) ArithmeticException
- (C) IOException
- (D) ClassCastException

Answer Key

Q	Ans	Q	Ans	Q	Ans
1	B	6	C	11	C
2	B	7	C	12	B
3	C	8	D	13	C
4	C	9	D	14	B
5	B	10	B	15	B

Table: Answer key for the 15 MCQs

Short explanations

- **Q1 (B):** start() creates a new thread and then calls run() in it.
- **Q2 (B):** A class that already extends a class implements Runnable, since Java has no multiple class inheritance.
- **Q3 (C):** Direct call of run() is an ordinary method call; no new thread is made.
- **Q4 (C):** Thread.NORM_PRIORITY is 5.
- **Q5 (B):** MIN_PRIORITY is 1 and MAX_PRIORITY is 10.
- **Q6 (C):** join() makes the caller wait for the target thread to die.
- **Q7 (C):** A thread that has already started cannot be restarted: IllegalStateException.
- **Q8 (D):** synchronized gives one thread at a time access to the lock.
- **Q9 (D):** A thread whose run() has finished is dead (terminated).
- **Q10 (B):** Throwable is the parent of Error and Exception.
- **Q11 (C):** IOException is checked; the others are unchecked subclasses of RuntimeException.
- **Q12 (B):** The first matching catch is executed, so specific exceptions go before general ones.
- **Q13 (C):** finally executes regardless of whether an exception occurs.
- **Q14 (B):** throw is used inside a method to throw an exception object; throws only declares.
- **Q15 (B):** Integer division by zero raises ArithmeticException ("/ by zero").

Broad Questions (10)

Answer the following questions in detail. Where applicable, write syntax, program and output.

- Q1.** What is a thread? Explain the two ways of creating a thread in Java with syntax and a suitable program for each.
- Q2.** Compare extending the Thread class with implementing the Runnable interface. Explain the difference between start() and run().
- Q3.** Explain how a thread can be stopped and how it can be blocked. Mention the deprecated methods and the recommended alternatives with an example.
- Q4.** Explain the life cycle of a thread with a neat diagram. Describe each state and the methods that cause transitions between them.
- Q5.** Describe the important methods of the Thread class (sleep, join, yield, isAlive, setName, setPriority, interrupt) with their purpose.
- Q6.** What are thread exceptions? Explain InterruptedException, IllegalThreadStateException, IllegalArgumentException and IllegalMonitorStateException with causes.
- Q7.** Explain thread priority. State the priority constants, the default value and how the scheduler uses priority. Write a program to set and display priorities.
- Q8.** What is synchronization? Explain the race condition and how synchronized methods and synchronized blocks solve it, with a program. Define deadlock.
- Q9.** Define an exception. Classify errors in Java and explain checked and unchecked exceptions with the exception hierarchy diagram.
- Q10.** Explain the use of try, multiple catch and finally with syntax and programs. How are exceptions (getMessage, printStackTrace, user-defined exceptions with throw and throws) used for debugging?

End of Chapter 4: Multithreaded Programming and Exception Handling