

JAVA EXCEPTIONS & MULTITHREADING

Exception Handling एवं Threads

सरल Hindi-English व्याख्या के साथ

12 अध्यायों में Unit III की व्यवस्थित शिक्षा

EXCEPTIONS

THREADS

MCQs

पुस्तक का परिचय

How to Use This Book

यह पुस्तक आपके Unit III syllabus के आधार पर exception handling, try-catch, nested try, throw/throws, finally, checked/unchecked exceptions, custom exceptions, thread model, lifecycle, priorities, synchronization और inter-thread communication को एक क्रम में प्रस्तुत करती है।

व्याख्या Hindi-English mix में है। Technical terms, Java syntax और commands English में रखी गई हैं। प्रत्येक अध्याय में explanations, examples, outputs, revision और अभ्यास दिए गए हैं।

पढ़ने का तरीका

पहले concept पढ़ें, फिर program स्वयं type करके चलाएँ। Output बदलने के लिए input values बदलें। MCQs हल करने के बाद answer key देखें। केवल code याद करने के स्थान पर execution की प्रक्रिया समझें।

Version और Program Conventions

अधिकांश examples traditional class-based Java syntax पर आधारित हैं। Examples Java 17-compatible platform-thread forms पर आधारित हैं। Legacy suspend/resume/stop को historical context में समझाया गया है; runnable examples cooperative control उपयोग करती हैं। Thread scheduling के कारण जहाँ output बदल सकता है, वहाँ possible/typical output बताया गया है।

Public class वाला program उसी class name की .java file में save करें। छोटे fragments किसी उचित method/class के भीतर उपयोग करें। अलग examples में समान helper class names हो सकते हैं; उन्हें अलग folders में रखें।

Symbols और Design

नीले boxes में code और commands, हल्के हरे tables में comparisons और पीले boxes में exam points दिए गए हैं। अध्याय 12 पूरे unit का revision, MCQ answer key और mini project प्रदान करता है।

विषय-सूची

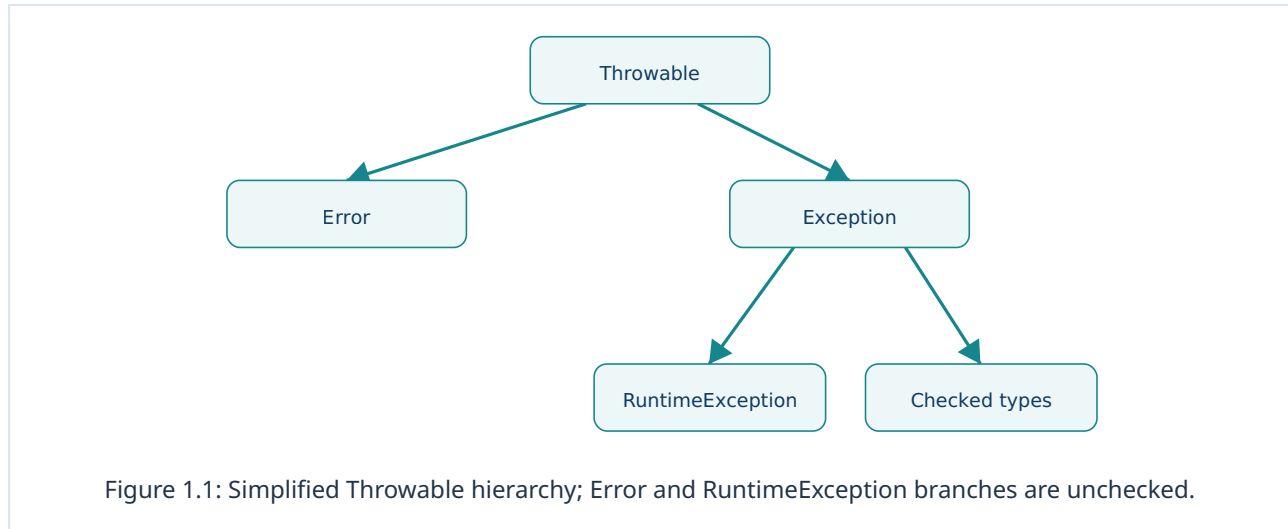
Contents / Chapters 1-12

01. Exceptions का परिचय	4
02. try-catch एवं Multiple Catch Blocks	7
03. Nested try, throw, throws एवं finally	11
04. Built-In, Checked एवं Unchecked Exceptions	14
05. Custom Exception Classes	17
06. Java Thread Model एवं Main Thread	20
07. Threads बनाना	23
08. Thread Life Cycle एवं Priorities	26
09. Synchronization एवं Race Conditions	29
10. Inter-Thread Communication	32
11. Thread Suspension, Resumption एवं Stopping	35
12. Revision, Question Bank एवं Mini Project	38

UNIT III / CHAPTER 01

अध्याय 1: Exceptions का परिचय

Abnormal Conditions एवं Throwable Hierarchy



1.1 Learning Objectives

Exception का अर्थ, normal और abrupt execution, Throwable hierarchy, stack trace तथा Exception और Error में अंतर समझेंगे।

1.2 Exception क्या है?

Exception ऐसी स्थिति को represent करती है जिससे execution का normal flow बाधित हो सकता है। Java में thrown condition को Throwable या उसकी subclass के object के माध्यम से represent किया जाता है। उदाहरण: zero से integer division, invalid array index या file का उपलब्ध न होना।

Exception handling का उद्देश्य हर गलती छिपाना नहीं है। Program उपयुक्त recovery, meaningful error message या failure propagation कर सकता है।

1.3 Error के विभिन्न अर्थ

शब्द	उदाहरण	पहचान
Compilation error	Missing semicolon	Compiler
Logical error	गलत formula	Output checking
Runtime exception	Invalid index	Execution
java.lang.Error	StackOverflowError	गंभीर runtime condition

“Error” सामान्य भाषा में किसी mistake का नाम है; java.lang.Error hierarchy का विशिष्ट type है। दोनों को समान न समझें।

1.4 Throwable Hierarchy

Throwable के प्रमुख direct branches Error और Exception हैं। RuntimeException, Exception की subclass है। Checked/unchecked classification hierarchy के आधार पर होती है।

Type	भूमिका	उदाहरण
Throwable	Thrown objects का root type	Base type
Error	सामान्यतः serious runtime failures	StackOverflowError
Exception	Application-level exceptional conditions	IOException
RuntimeException	Unchecked exception branch	NullPointerException

1.5 Exception के बिना Recovery

```
public class UnhandledDemo {
    public static void main(String[] args) {
        int divisor = 0;
        System.out.println(10 / divisor);
        System.out.println("This line is not reached");
    }
}
```

Integer division ArithmeticException throw करती है। अंतिम line नहीं चलती। Uncaught exception current thread के termination तक पहुँच सकती है। अन्य threads होने पर पूरा process तत्काल बंद होना आवश्यक नहीं।

1.6 Stack Trace

Stack trace failure के call path की जानकारी देती है। उसमें exception type, message, methods और source locations मिल सकते हैं। Line numbers compilation/debug metadata पर निर्भर हैं।

```
java.lang.ArithmeticException: / by zero
    at UnhandledDemo.main(UnhandledDemo.java:4)
```

यह illustrative trace है; actual formatting और line number environment के अनुसार बदल सकते हैं। सबसे पहले exception type पढ़ें, फिर application की relevant location जाँचें।

1.7 Handling का उदाहरण

```
public class BasicExceptionDemo {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            System.out.println(10 / divisor);
        } catch (ArithmeticException exception) {
```

```

        System.out.println("Cannot divide an integer by zero.");
    }
    System.out.println("Program continues.");
}
}

```

Output:

```

Cannot divide an integer by zero.
Program continues.

```

Thrown condition matching catch में handle हुई, इसलिए try-catch के बाद execution जारी रहा।

1.8 Propagation

यदि current method suitable catch नहीं देती, तो exception caller की ओर propagate होती है। Call stack में matching handler खोजी जाती है। Unwinding के दौरान applicable finally blocks और synchronized cleanup semantics लागू होती हैं।

```

static int divide(int a, int b) {
    return a / b;
}

```

Caller divide(10,0) से आई ArithmeticException handle कर सकता है। Method हर exception स्वयं handle करना आवश्यक नहीं।

1.9 Good Practices

- Expected recoverable conditions को सही level पर handle करें।
- Empty catch से failure silently न छिपाएँ।
- Invalid input और programmer bug में अंतर करें।
- Throwable या Error को सामान्य recovery catch के रूप में अंधाधुंध न पकड़ें।
- Meaningful messages और causes सुरक्षित रखें।

1.10 Revision एवं अभ्यास

Exceptions runtime पर occur होती हैं। Checked classification compiler को catch-or-declare obligation बताती है; इसका अर्थ exception compile time पर घटती है नहीं। Uncaught failure thread-specific हो सकती है।

अभ्यास: Integer division का handled program बनाएँ। Stack trace में type और source location पहचानें।

MCQs: 1. Thrown object root: A. Object only B. Throwable C. Thread D. Runnable 2.

RuntimeException किसकी subclass? A. Exception B. Error C. String D. Thread 3. Logical error हमेशा exception देती है? A. हाँ B. नहीं C. केवल int D. केवल static

उत्तर: 1-B, 2-A, 3-B.

UNIT III / CHAPTER 02

अध्याय 2: try-catch एवं Multiple Catch Blocks

Matching Handlers एवं Multi-Catch

2.1 Learning Objectives

Try में protected operations, catch matching, catch order, multiple handlers और multi-catch syntax समझेंगे।

2.2 Basic Syntax

```
try {  
    // Operations that may throw  
} catch (ExceptionType exception) {  
    // Handling action  
}
```

Try body में exception आते ही उसके बाकी statements skip होते हैं। Matching catch execute होती है। Catch के बाद try block वहीं से resume नहीं होता जहाँ failure हुई थी।

2.3 Complete Example

```
public class CatchFlowDemo {  
    public static void main(String[] args) {  
        int divisor = 0;  
        try {  
            System.out.println("Before division");  
            System.out.println(10 / divisor);  
            System.out.println("After division");  
        } catch (ArithmeticException exception) {  
            System.out.println("Division failed");  
        }  
        System.out.println("After try-catch");  
    }  
}
```

Output:

```
Before division  
Division failed  
After try-catch
```

2.4 Multiple Catch Blocks

```
public class MultipleCatchDemo {
    static void process(String text) {
        try {
            int index = Integer.parseInt(text);
            int[] values = {10, 20};
            System.out.println(values[index]);
        } catch (NumberFormatException exception) {
            System.out.println("Enter a valid integer.");
        } catch (ArrayIndexOutOfBoundsException exception) {
            System.out.println("Index is outside the array.");
        }
    }
    public static void main(String[] args) {
        process("abc");
        process("5");
        process("1");
    }
}
```

Output:

```
Enter a valid integer.
Index is outside the array.
20
```

एक thrown exception के लिए इस try के catch chain से पहला matching catch चुना जाता है। सभी catches एक साथ नहीं चलतीं।

2.5 Catch Order

Specific exception handlers broader superclass handler से पहले रखें।

```
try {
    Integer.parseInt("abc");
} catch (NumberFormatException exception) {
    System.out.println("Invalid number");
} catch (RuntimeException exception) {
    System.out.println("Other runtime problem");
}
```

NumberFormatException पहले RuntimeException catch रख देने पर subsequent specific catch unreachable होगी।

2.6 Multi-Catch

Java 7 से एक handler कई unrelated alternative types handle कर सकता है।

```
try {
    int[] values = {10, 20};
    int index = Integer.parseInt("5");
    System.out.println(values[index]);
} catch (NumberFormatException |
        ArrayIndexOutOfBoundsException exception) {
    System.out.println("Invalid number or index.");
}
```

यह multiple separate catch blocks से अलग syntax है। Same handling logic हो तो duplication घटा सकती है।

2.7 Multi-Catch Restrictions

Alternative types में subclass-superclass relationship नहीं हो सकती।

```
// INVALID alternatives:
// catch (NumberFormatException | RuntimeException e) { }
```

Multi-catch parameter implicitly final है; उसे नया exception reference assign नहीं कर सकते।

2.8 Catch Parameter का उपयोग

Operation	काम
exception.getMessage()	Message, null भी हो सकती है
exception.toString()	Type और message representation
exception.printStackTrace()	Stack trace output
exception.getCause()	Underlying cause, यदि उपलब्ध

User-facing output में implementation details तभी दिखाएँ जब उपयोगी हों। Development के लिए meaningful trace/logging रखें।

2.9 Checked Handler की Reachability

Compiler checked exception catches के लिए analysis करता है। ऐसा checked type catch करना जो try body से सम्भव नहीं, compilation error हो सकता है; Exception/Throwable catches के विशेष language rules भी हैं। इसे “कोई भी catch कहीं भी लिख सकते हैं” न समझें।

2.10 Common Mistakes

गलती	प्रभाव
Broad catch पहले	Specific catch unreachable
Empty catch	Failure silently hidden

गलती	प्रभाव
Related multi-catch alternatives	Compilation error
Catch को retry मानना	Failed statement अपने आप दोबारा नहीं चलती
हर failure पर generic success print	गलत application behavior

2.11 अभ्यास एवं MCQs

अभ्यास: Number parsing और array indexing के लिए अलग catches लिखें। फिर same handling को multi-catch में बदलें।

1. Same thrown exception के लिए matching catch count: A. सभी B. पहला matching C. हमेशा दो D. कोई नहीं
2. Specific catch का क्रम: A. Broad के बाद B. Broad से पहले C. कोई नियम नहीं D. केवल finally में
3. Multi-catch alternatives related हों: A. Valid B. Invalid C. Required D. Always runtime error

उत्तर: 1-B, 2-B, 3-B.

UNIT III / CHAPTER 03

अध्याय 3: Nested try, throw, throws एवं finally

Propagation और Cleanup

3.1 Nested try

एक try के भीतर दूसरी try हो, तो inner और outer handlers अलग levels पर exceptions handle कर सकते हैं।

```
public class NestedTryDemo {
    public static void main(String[] args) {
        try {
            try {
                Integer.parseInt("abc");
            } catch (NumberFormatException exception) {
                System.out.println("Inner handler");
            }
            int[] values = {10};
            System.out.println(values[2]);
        } catch (ArrayIndexOutOfBoundsException exception) {
            System.out.println("Outer handler");
        }
    }
}
```

Output: Inner handler, फिर Outer handler। Inner failure handle हो जाने के बाद outer try के अगले statements चलते हैं।

3.2 throw Keyword

Throw actual Throwable object को exceptional control flow में भेजता है।

```
static void validateAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Minimum age is 18.");
    }
}
```

यह condition true होने पर नया exception throw करता है। Throw के बाद उसी normal path के statements execute नहीं होते।

3.3 throws Clause

Throws method/constructor declaration में सम्भव exception types बताती है। Checked exception के लिए catch-or-declare obligation caller तक propagate हो सकता है।

```
static void pause() throws InterruptedException {
    Thread.sleep(10);
}
```

Throws लिखने से exception अपने आप throw या handle नहीं होती। Unchecked types declare करना optional है।

3.4 throw बनाम throws

आधार	throw	throws
स्थान	Body में statement	Method declaration
क्या देता है?	Exception object/expression	Exception type list
कार्य	वास्तव में throw	सम्भावना/contract declare
उदाहरण	throw new IOException()	throws IOException

3.5 finally Block

Finally normal completion और exception-related abrupt completion के दौरान cleanup कर सकती है।

```
public class FinallyDemo {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            System.out.println("Try begins");
            System.out.println(10 / divisor);
        } catch (ArithmeticException exception) {
            System.out.println("Catch executes");
        } finally {
            System.out.println("Finally executes");
        }
        System.out.println("After try statement");
    }
}
```

Output: Try begins; Catch executes; Finally executes; After try statement।

3.6 Finally की सीमाएँ

“Finally हर परिस्थिति में अवश्य चलती है” पूर्णतः सही नहीं है। Process/JVM termination, abrupt external kill या execution कभी finally तक न पहुँचना जैसी conditions इसे रोक सकती हैं। System.exit के बाद ordinary pending finally blocks पर भरोसा न करें।

3.7 Return और finally

```
static int value() {
    try {
        return 10;
    } finally {
        System.out.println("Cleanup");
    }
}
```

Return expression evaluate होती है, finally execute होती है, फिर method result देती है यदि finally normally complete करे।

Finally में return या नया throw prior return/exception को replace कर सकता है। इसलिए finally से return करना सामान्यतः खराब practice है।

```
// Avoid this design:
// try { return 10; } finally { return 20; }
```

इस case में result 20 होगा और earlier return दब जाती है।

3.8 try-finally और try-with-resources

Ordinary try को catch, finally या दोनों चाहिए। Try-with-resources में resource header और appropriate AutoCloseable resource के साथ catch/finally optional हो सकते हैं।

```
import java.io.StringReader;
public class ResourceDemo {
    public static void main(String[] args) throws Exception {
        try (StringReader reader = new StringReader("Java")) {
            System.out.println((char) reader.read());
        }
    }
}
```

Output: J। AutoCloseable resources scope समाप्त होने पर close होती हैं। Multiple resources reverse declaration order में close होती हैं। Body और close दोनों throw करें, तो close exception suppressed हो सकती है।

3.9 Revision एवं अभ्यास

Throw object भेजता है; throws types declare करती है। Finally cleanup के लिए है, failure छिपाने के लिए नहीं। Resource management में try-with-resources उपयोगी है।

अभ्यास: Age validation exception catch करें। Nested try example बनाइए। Return/finally flow समझाइए।

MCQs: 1. Actual exception throw: A. throws B. throw C. finally D. import 2. Declaration clause: A. throws B. throw C. catch only D. new 3. Finally में return: A. Earlier return override कर सकती है B. हमेशा invalid C. कभी नहीं चलेगी D. Thread बनाती है

उत्तर: 1-B, 2-A, 3-A.

UNIT III / CHAPTER 04

अध्याय 4: Built-In, Checked एवं Unchecked Exceptions

Classification और Catch-or-Declare Rule

4.1 Learning Objectives

Hierarchy से exception category पहचानेंगे। Checked compile-time obligations और runtime occurrence के बीच अंतर स्पष्ट करेंगे।

4.2 Checked Exceptions

Throwable hierarchy में RuntimeException और उसकी subclasses तथा Error और उसकी subclasses को छोड़कर बाकी exception types checked classification में आती हैं। Common examples IOException और InterruptedException हैं।

Compiler सम्भव checked exception को suitable catch या method throws clause से account करने की माँग करता है। Checked exception **runtime पर** occur होती है; checking obligation compile time पर लागू होती है।

4.3 Unchecked Exceptions

RuntimeException branch और Error branch unchecked हैं। इन्हें catch या declare करना compile-time requirement नहीं है। Appropriate handling या prevention फिर भी आवश्यक हो सकती है।

4.4 प्रमुख Built-In Types

Type	Category	सामान्य कारण
ArithmeticException	Unchecked	Integer zero divisor
NullPointerException	Unchecked	Null पर instance operation
ArrayIndexOutOfBoundsException	Unchecked	Invalid array index
NumberFormatException	Unchecked	Invalid numerical text
ClassCastException	Unchecked	Incompatible runtime cast
IllegalArgumentException	Unchecked	Invalid argument
IOException	Checked	I/O failure
FileNotFoundException	Checked	File opening condition
InterruptedException	Checked	Interruptible waiting interrupted

Type	Category	सामान्य कारण
StackOverflowError	Unchecked Error	अत्यधिक recursive stack use

4.5 Integer और Floating Division

```
int divisor = 0;
// 10 / divisor throws ArithmeticException
System.out.println(10.0 / 0.0); // Infinity
System.out.println(0.0 / 0.0); // NaN
```

Floating division by zero integer division जैसा exception behavior नहीं देती।

4.6 Checked Exception का Complete Example

```
import java.io.IOException;

public class CheckedDemo {
    static void load(boolean available) throws IOException {
        if (!available) {
            throw new IOException("Data is unavailable");
        }
        System.out.println("Data loaded");
    }
    public static void main(String[] args) {
        try {
            load(false);
        } catch (IOException exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

Output: Data is unavailable! Throws declaration caller की obligation स्पष्ट करती है।

4.7 Prevention बनाम Handling

Array bounds validate करना अक्सर exception-driven normal control flow से बेहतर है। Null contract और argument validation API behavior स्पष्ट करते हैं। File operations में pre-check करने के बाद भी environment बदल सकता है, इसलिए actual operation की IOException handle करनी पड़ सकती है।

4.8 Overriding और Checked Exceptions

Overriding method superclass method से broader new checked exception declare नहीं कर सकती। Same या narrower allowed checked types declare कर सकती है, या checked exception छोड़ सकती है। Unchecked exception declarations पर यही widening restriction नहीं लगती।

4.9 Comparison

आधार	Checked	Unchecked
Compiler catch-or-declare	हाँ, applicable analysis	नहीं
Runtime occurrence	हाँ	हाँ
Example	IOException	NumberFormatException
हमेशा recoverable?	नहीं	नहीं
हमेशा programmer bug?	नहीं	नहीं

Category केवल recovery likelihood से तय नहीं होती; type hierarchy निर्णायक है।

4.10 Revision एवं अभ्यास

Exception superclass checked है, लेकिन RuntimeException unchecked branch है। Catch(Exception) Error branch को नहीं पकड़ती। Application को every Throwable blindly recover नहीं करना चाहिए।

अभ्यास: पाँच built-in exceptions की conditions लिखें। CheckedDemo से catch हटाकर throws declaration का प्रभाव समझें।

MCQs: 1. Checked type: A. IOException B. NullPointerException C. NumberFormatException D. ArithmeticException 2. Error branch: A. Checked B. Unchecked C. String D. Runnable 3. Checked exception occur होती है: A. केवल compile time B. Runtime पर C. केवल import में D. केवल JAR बनाते समय

उत्तर: 1-A, 2-B, 3-B.

UNIT III / CHAPTER 05

अध्याय 5: Custom Exception Classes

Domain-Specific Failures

5.1 Custom Exception क्या है?

Application की specific failure को meaningful type देने के लिए programmer custom exception बना सकता है। Example: InvalidMarksException, InsufficientBalanceException या AdmissionException।

Checked custom type सामान्यतः Exception extend करती है। Unchecked custom type RuntimeException extend कर सकती है। Choice API contract और failure handling design के अनुसार करें।

5.2 Checked Custom Exception

```
class InvalidMarksException extends Exception {
    private static final long serialVersionUID = 1L;
    InvalidMarksException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    static void validate(int marks) throws InvalidMarksException {
        if (marks < 0 || marks > 100) {
            throw new InvalidMarksException("Marks must be 0-100.");
        }
        System.out.println("Valid marks: " + marks);
    }
    public static void main(String[] args) {
        try {
            validate(120);
        } catch (InvalidMarksException exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

Output: Marks must be 0-100। serialVersionUID serialization-related identifier है; basic handling का keyword नहीं।

5.3 Constructors और Message

Superclass constructor को message देने से getMessage() meaningful information लौटाती है। Cause preserve करने के लिए:

```
class DataLoadException extends Exception {
    private static final long serialVersionUID = 1L;
    DataLoadException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Wrapping करते समय original cause बचाना debugging में सहायता करता है।

5.4 Unchecked Custom Exception

```
class InvalidAgeException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
    InvalidAgeException(String message) {  
        super(message);  
    }  
}
```

इस type के लिए catch-or-declare compile-time obligation नहीं है। इसका अर्थ failure ignore करना उपयुक्त है नहीं।

5.5 Custom Type कब उपयोग करें?

जब caller को specific domain condition पहचानकर अलग response देना हो, custom type उपयोगी है। सामान्य invalid argument के लिए existing IllegalArgumentException पर्याप्त हो सकती है। हर छोटी condition के लिए नया exception class बनाना आवश्यक नहीं।

5.6 Exception Propagation और Recovery

Validation method throw करती है; UI layer user से correction माँग सकती है; business layer transaction abort कर सकती है। Handling responsibility सही abstraction level पर रखें। Exception message और success message को अलग behavior के रूप में design करें।

5.7 Bank Example का Method Fragment

```
static void withdraw(double balance, double amount)  
    throws InsufficientBalanceException {  
    if (amount > balance) {  
        throw new InsufficientBalanceException("Insufficient balance");  
    }  
}
```

इस fragment के लिए InsufficientBalanceException class अलग declare करनी होगी। Full application में negative amounts, representation और state update भी design करने होंगे। Fragment स्वयं balance update नहीं करता।

5.8 Good Practices

- Exception name failure condition बताए।
- Useful message दें; sensitive data अनावश्यक न डालें।
- Cause खोकर नया generic message न फेंकें।
- Exceptions normal successful iteration का विकल्प नहीं हैं।
- Unchecked choice केवल compiler obligation बचाने के लिए न करें।

5.9 Quick Revision एवं अभ्यास

Exception extend करने पर generally checked custom type; RuntimeException extend करने पर unchecked type। Super(message) message initialize करता है। Throw और throws दोनों required roles अलग हैं।

अभ्यास: InvalidAgeException और InsufficientBalanceException बनाएँ। Valid तथा invalid paths दोनों का output दिखाएँ।

MCQs: 1. Checked custom base: A. Exception B. Thread C. String D. Runnable 2. Message constructor call: A. super(message) B. import(message) C. throws(message) D. final(message) 3. RuntimeException subclass: A. Unchecked B. Checked हमेशा C. Interface D. Error की direct subclass

उत्तर: 1-A, 2-A, 3-A.

UNIT III / CHAPTER 06

अध्याय 6: Java Thread Model एवं Main Thread

Concurrency, Processes और Threads

6.1 Thread क्या है?

Thread program के execution का एक क्रम है। Process में अनेक threads हो सकती हैं। Threads heap और process resources साझा कर सकती हैं, लेकिन प्रत्येक की execution stack तथा program-counter state अलग होती है।

6.2 Process बनाम Thread

आधार	Process	Thread
भूमिका	Running application environment	Execution path
Address space	सामान्यतः अलग process space	Process के भीतर shared
Communication	IPC आदि	Shared objects और coordination
Failure impact	Process-specific	Shared application state प्रभावित हो सकती है

Thread केवल “छोटा program” नहीं है; shared state के कारण concurrency design आवश्यक है।

6.3 Concurrency और Parallelism

Concurrency में tasks progress overlap कर सकती हैं। Parallelism में tasks वास्तव में एक समय पर अलग execution resources पर चलती हैं। Single-core system scheduling से concurrent progress दे सकता है; true CPU parallelism के लिए उपयुक्त hardware चाहिए।

6.4 Java Thread Model

इस पुस्तक में conventional platform Thread objects और monitors का उपयोग है। Thread class control methods देती है; Runnable task contract देती है। Newer virtual-thread APIs इस basic unit के examples में उपयोग नहीं की गई हैं।

Thread lifecycle और scheduling JVM तथा underlying system से संबंधित हैं। Program को arbitrary execution order पर निर्भर नहीं करना चाहिए।

6.5 Main Thread

Traditional launcher main method को initial application thread पर चलाता है, जिसका नाम सामान्यतः main होता है। नाम fixed language guarantee नहीं; code name बदल सकता है।

```

public class MainThreadDemo {
    public static void main(String[] args) {
        Thread current = Thread.currentThread();
        System.out.println("Name: " + current.getName());
        System.out.println("Priority: " + current.getPriority());
        System.out.println("State: " + current.getState());
    }
}

```

Typical output:

```

Name: main
Priority: 5
State: RUNNABLE

```

Current executing thread RUNNABLE classification में होती है। Priority/default inheritance configuration के अनुसार बदल सकती है।

6.6 प्रमुख Methods

Method	काम
Thread.currentThread()	Current Thread reference
getName()/setName()	Name पढ़ना/बदलना
start()	New execution schedule करना
run()	Task body
join()	Target termination की प्रतीक्षा
sleep()	Current thread को timed sleep
interrupt()	Interruption request
isAlive()	Started but not terminated?

6.7 User और Daemon Threads

Platform-thread model में daemon status JVM termination behavior से संबंधित है। Non-daemon threads समाप्त होने पर JVM daemon work के पूरा होने की guarantee नहीं देती। Important data saving के लिए daemon thread completion पर निर्भर न रहें। Daemon flag thread start होने से पहले set करना चाहिए।

6.8 Shared State

दो threads यदि shared counter update करें, तो unsynchronized increment में lost updates हो सकती हैं। Local variables अलग stacks में होना shared referenced object को अपने आप thread-safe नहीं बनाता। Synchronization chapter इस समस्या का समाधान देता है।

6.9 Revision एवं अभ्यास

Threads share heap, individual stacks रखती हैं। Scheduling order guaranteed नहीं है। Main thread और worker threads application design में सहयोग करती हैं। Sleep thread creation नहीं है।

अभ्यास: Current thread की details print करें। Concurrency और parallelism का अंतर उदाहरण सहित लिखें।

MCQs: 1. Current thread reference: A. `currentThread()` B. `newCurrent()` C. `mainThread()` D. `run()`
2. Threads सामान्यतः क्या share करती हैं? A. Heap B. एक ही stack हमेशा C. एक PC हमेशा D. Local primitive variables हमेशा
3. Concurrency का अर्थ हमेशा parallel CPU execution? A. हाँ B. नहीं C. केवल Java में हाँ D. केवल main

उत्तर: 1-A, 2-A, 3-B.

UNIT III / CHAPTER 07

अध्याय 7: Threads बनाना

Extending Thread एवं Implementing Runnable

7.1 Learning Objectives

Thread subclass और Runnable task से execution शुरू करना सीखेंगे। Start/run के अंतर, join और uncaught failures को समझेंगे।

7.2 Thread Class Extend करना

```
class Worker extends Thread {
    Worker(String name) {
        super(name);
    }
    @Override
    public void run() {
        System.out.println("Worker: " + getName());
    }
}

public class ThreadSubclassDemo {
    public static void main(String[] args) throws InterruptedException {
        Worker worker = new Worker("Worker-1");
        worker.start();
        worker.join();
        System.out.println("Main finished");
    }
}
```

Output:

```
Worker: Worker-1
Main finished
```

Join के कारण main की अंतिम line worker termination के बाद है।

7.3 Runnable Implement करना

```
class Task implements Runnable {
    @Override
    public void run() {
        System.out.println("Task thread: "
            + Thread.currentThread().getName());
    }
}

public class RunnableDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(new Task(), "Task-1");
    }
}
```

```

        worker.start();
        worker.join();
        System.out.println("Done");
    }
}

```

Output: Task thread: Task-1; Done | Runnable separation से task और execution control अलग रहते हैं। Class किसी अन्य superclass को extend करते हुए Runnable implement कर सकती है।

7.4 start() बनाम run()

Method	व्यवहार
start()	New thread execution schedule; JVM run call करती है
run() directly	Ordinary method call; current thread execute करती है

```

Thread worker = new Thread(() ->
    System.out.println(Thread.currentThread().getName()));
worker.run(); // Current calling thread

```

Lambda syntax Runnable task का concise रूप है। Direct run कोई नया execution thread नहीं बनाती।

7.5 Thread Restart नहीं होती

Thread object को केवल एक बार start कर सकते हैं। Termination के बाद उसी object पर फिर start IllegalThreadStateException दे सकती है। दूसरा execution चाहिए तो नया Thread बनाइए।

7.6 दो Threads का Example

```

public class TwoThreadsDemo {
    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println(Thread.currentThread().getName()
                    + ": " + i);
            }
        };
        Thread first = new Thread(task, "A");
        Thread second = new Thread(task, "B");
        first.start();
        second.start();
        first.join();
        second.join();
        System.out.println("Both finished");
    }
}

```

एक possible output:

```
A: 1
B: 1
A: 2
B: 2
A: 3
B: 3
Both finished
```

A/B lines का interleaving बदल सकता है। प्रत्येक thread अपनी loop में 1,2,3 क्रम रखती है। Both finished दोनों joins के बाद है।

7.7 join और Failure Propagation

Join target thread की termination की प्रतीक्षा करती है; यह worker की thrown exception को automatically caller में rethrow नहीं करती। UncaughtExceptionHandler या task-result abstractions से worker failures report की जा सकती हैं।

7.8 Comparison

आधार	Thread subclass	Runnable
Declaration	extends Thread	implements Runnable
Other superclass	सीधे नहीं	संभव
Task/control separation	कम	स्पष्ट
Execution	object.start()	new Thread(task).start()

7.9 Revision एवं अभ्यास

Run task body है। Start concurrent execution शुरू करती है। Join completion wait है, cancellation नहीं। Worker failure main try-catch में केवल start/join के आसपास रखने से automatically catch नहीं होती।

अभ्यास: दो threads even और odd numbers print करें। Join के बाद completion message दें।

MCQs: 1. New execution शुरू करती है: A. run directly B. start C. getName D. sleep 2. Thread object restart: A. हमेशा B. permitted नहीं C. केवल daemon D. priority 10 पर 3. Join क्या करती है? A. Target termination wait B. दो threads merge C. Exception हमेशा rethrow D. Force stop

उत्तर: 1-B, 2-B, 3-A.

UNIT III / CHAPTER 08

अध्याय 8: Thread Life Cycle एवं Priorities

States, sleep, yield एवं Scheduling

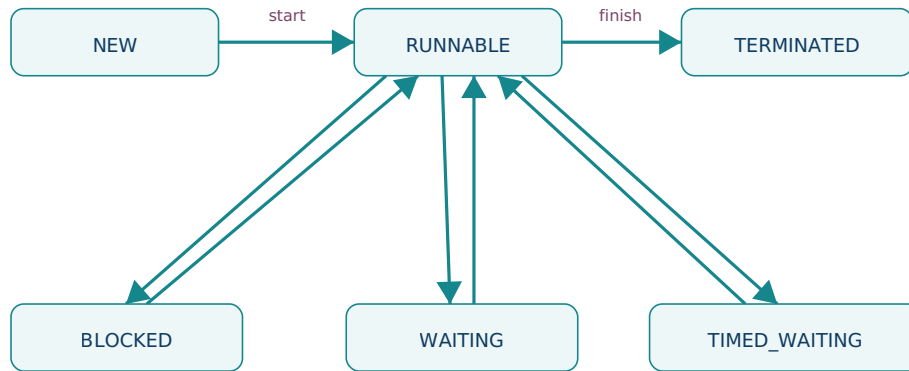


Figure 8.1: Selected lifecycle transitions. Waiting states return toward RUNNABLE; no separate RUNNING enum.

8.1 Thread States

Java Thread.State में छह values हैं। Textbook “running” को अलग conceptual stage दिखा सकती है, लेकिन Java enum में अलग RUNNING state नहीं है।

State	अर्थ
NEW	Thread बनी, start नहीं
RUNNABLE	JVM में runnable/executing; OS scheduling का इंतजार भी संभव
BLOCKED	Intrinsic monitor lock acquire करने की प्रतीक्षा
WAITING	Untimed wait/join जैसी waiting
TIMED_WAITING	Sleep/timed wait/join जैसी waiting
TERMINATED	Run समाप्त या uncaught failure से समाप्त

State snapshot बदल सकती है। getState() coordination protocol का विकल्प नहीं है।

8.2 Deterministic State Example

```
public class ThreadStateDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> { });
        System.out.println(worker.getState());
        worker.start();
        worker.join();
        System.out.println(worker.getState());
    }
}
```

Output: NEW और TERMINATED। Start के तुरंत बाद state print करने पर result scheduling पर निर्भर होगा।

8.3 sleep()

```
Thread.sleep(100);
```

Current thread timed sleep करती है। Time millisecond request है; exact wake-up instant timers/scheduler पर निर्भर है। Sleep held monitors को release नहीं करती।

Sleep को shared-memory visibility या दूसरे task की completion का प्रमाण न मानें। Completion के लिए join या appropriate coordination उपयोग करें।

8.4 yield()

Thread.yield() scheduler को hint देती है कि current thread processor use छोड़ने को willing है। Scheduler hint ignore कर सकता है। Yield guaranteed context switch, fairness या synchronization नहीं है।

8.5 Thread Priorities

Platform Thread priority constants:

Constant	Value
MIN_PRIORITY	1
NORM_PRIORITY	5
MAX_PRIORITY	10

Priority integer range में होनी चाहिए। Effective scheduling system और thread-group limits पर निर्भर हो सकती है। Higher priority पहले completion की guarantee नहीं देती।

8.6 Priority Example

```
public class PriorityDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() ->
            System.out.println("Worker executed"));
        worker.setPriority(Thread.NORM_PRIORITY);
        System.out.println("Configured priority: " + worker.getPriority());
        worker.start();
        worker.join();
    }
}
```

Typical output: Configured priority: 5; Worker executed। यह example priority के कारण ordering साबित नहीं करता; first line start से पहले है।

8.7 Waiting, Blocked और Sleeping

Intrinsic lock acquisition के लिए BLOCKED है। Object.wait() condition notification की प्रतीक्षा देती है। Sleep timed delay है। सभी blocking mechanisms Thread.State.BLOCKED ही दिखाएँ, यह सही नहीं; library coordination mechanisms अलग states उपयोग कर सकते हैं।

8.8 Interruption

Interrupt request thread को cooperate करने का संकेत है। Sleep/wait/join InterruptedException throw कर सकती हैं और interrupt status clear होता है। IsInterrupted() status पढ़ती है; Thread.interrupted() current thread का status पढ़कर clear करती है।

8.9 Revision एवं अभ्यास

No separate RUNNING enum। Priority order guarantee नहीं। Sleep monitor release नहीं करती। GetState snapshot है। Terminated Thread फिर NEW नहीं बनती।

अभ्यास: Start से पहले और join के बाद state print करें। Priority constants display करें। Sleep और wait की तुलना लिखें।

MCQs: 1. Thread.State में कौन नहीं? A. NEW B. RUNNING C. WAITING D. TERMINATED 2. sleep held monitor छोड़ती है? A. हाँ B. नहीं C. हमेशा आधे D. Priority पर निर्भर 3. Higher priority पहले finish guarantee? A. हाँ B. नहीं C. केवल 10 पर D. हमेशा Linux पर

उत्तर: 1-B, 2-B, 3-B.

UNIT III / CHAPTER 09

अध्याय 9: Synchronization एवं Race Conditions

Mutual Exclusion, Visibility एवं Locks

9.1 Race Condition

Shared state पर operations का outcome uncontrolled interleaving पर निर्भर हो, तो race condition पैदा हो सकती है। Counter++ read, add और write का compound operation है; यह ordinary variable पर atomic नहीं है।

यदि दो threads value 5 पढ़ें और दोनों 6 लिखें, तो दो increments के बाद expected 7 के स्थान पर 6 मिल सकता है। इसे lost update कहते हैं।

9.2 synchronized Keyword

Synchronized block या method intrinsic monitor lock का उपयोग करती है। एक monitor को एक समय पर एक thread own कर सकती है। Same lock पर mutually exclusive operations shared invariant सुरक्षित रख सकती हैं।

```
synchronized (lock) {  
    // Critical section  
}
```

Lock non-null object होना चाहिए। Null monitor expression NullPointerException देता है।

9.3 Instance और Static Lock

Form	Lock
synchronized instance method	this object monitor
static synchronized method	Declaring Class object monitor
synchronized(lock) block	दिए object का monitor

दो अलग Counter instances के synchronized methods सामान्यतः अलग locks लेते हैं। केवल synchronized keyword लगाने से सभी objects globally serialize नहीं होते।

9.4 Complete Counter Program

```
class Counter {
    private int value;
    synchronized void increment() {
        value++;
    }
    synchronized int getValue() {
        return value;
    }
}

public class SynchronizedCounterDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        };
        Thread first = new Thread(task);
        Thread second = new Thread(task);
        first.start();
        second.start();
        first.join();
        second.join();
        System.out.println(counter.getValue());
    }
}
```

Output: 20000। दोनों threads एक Counter और एक monitor साझा करती हैं। Joins completion सुनिश्चित करते हैं।

9.5 Visibility और Happens-Before

Synchronization केवल mutual exclusion नहीं; memory visibility में भी भूमिका रखती है। Monitor unlock का subsequent same-monitor lock से happens-before संबंध होता है। Thread start और successful join completion के भी visibility guarantees हैं।

Sleep का उपयोग memory barrier के विकल्प के रूप में नहीं करें। Shared updates की correctness proper synchronization mechanism से तय करें।

9.6 volatile का अंतर

Volatile variable reads/writes visibility और ordering guarantees देती हैं। लेकिन volatile int पर counter++ + atomic compound operation नहीं बनती।

```
private volatile boolean stopRequested;
```

Simple cooperative flag के लिए उपयोगी हो सकता है। Compound invariant के लिए lock या suitable atomic abstraction चाहिए।

9.7 Reentrancy

Intrinsic monitors reentrant हैं: thread जिस monitor को own करती है, उसे nested synchronized operation में फिर acquire कर सकती है। यह स्वयं से lock लेने पर automatic deadlock रोकता है; multiple-lock cycles फिर भी deadlock पैदा कर सकती हैं।

9.8 Deadlock

Thread A lock1 लेकर lock2 की प्रतीक्षा करे, और B lock2 लेकर lock1 की प्रतीक्षा करे, तो circular wait हो सकती है। Consistent lock acquisition order deadlock risk घटाने का प्रमुख तरीका है।

इस पुस्तक में जानबूझकर indefinitely hanging deadlock program execute नहीं किया गया है। Concept lock relationships से समझें और bounded diagnostics उपयोग करें।

9.9 Good Practices

- Related shared operations के लिए same lock रखें।
- Critical sections को आवश्यक सीमा तक रखें।
- Unknown callbacks या slow operations lock के भीतर सावधानी से करें।
- Thread.sleep lock नहीं छोड़ती।
- Exception से synchronized block छोड़ने पर monitor automatically release होता है।

9.10 Revision एवं अभ्यास

Synchronized protection तभी प्रभावी जब competing accesses same protocol follow करें। Different locks same state की mutual exclusion नहीं देते। Volatile increment atomic नहीं है।

अभ्यास: Shared counter synchronized और unsynchronized versions तुलना करें। Unsynchronized failure हर run में आवश्यक नहीं।

MCQs: 1. Instance synchronized lock: A. this B. हमेशा Class C. Thread name D. Package 2. Volatile counter++ atomic? A. हाँ B. नहीं C. केवल int D. केवल priority 10 3. Same monitor ownership: A. एक समय पर एक thread B. सभी C. दो हमेशा D. unlimited

उत्तर: 1-A, 2-B, 3-A.

UNIT III / CHAPTER 10

अध्याय 10: Inter-Thread Communication

wait(), notify() एवं notifyAll()

10.1 Learning Objectives

Condition-based waiting, monitor ownership, wait sets और producer-consumer coordination समझेंगे। Notifications तथा condition predicates का संबंध स्पष्ट करेंगे।

10.2 Object Methods

Wait, notify और notifyAll Object की methods हैं। Calling thread को संबंधित object का monitor own करना चाहिए; अन्यथा IllegalMonitorStateException हो सकती है।

Method	कार्य
wait()	Object wait set में wait; उस monitor को release
wait(timeout)	Time-bounded wait request
notify()	एक waiting thread चुनकर notify
notifyAll()	सभी wait-set threads notify

Notify के बाद waiting thread तुरंत execute या lock own नहीं कर लेती। उसे notifier के monitor छोड़ने के बाद lock reacquire करना पड़ता है।

10.3 while में Condition Check

```
synchronized (lock) {
    while (!condition) {
        lock.wait();
    }
    // Condition-dependent action
}
```

Spurious wakeups सम्भव हैं। Notification के बाद दूसरी thread condition बदल सकती है। इसलिए if के बजाय while से predicate दोबारा जाँचें।

10.4 wait और sleep का अंतर

आधार	wait	sleep
Declaring type	Object	Thread
Monitor ownership	Required	Required नहीं

आधार	wait	sleep
Lock release	संबंधित object का monitor	Held monitors नहीं
Purpose	Condition coordination	Timed delay
Wake reasons	Notification, interruption, timeout/spurious	Timer/interruption

Wait केवल उसी object का monitor release करती है; thread के अन्य held monitors नहीं।

10.5 Single-Slot Producer-Consumer

```

class Slot {
    private int value;
    private boolean full;

    synchronized void put(int newValue) throws InterruptedException {
        while (full) {
            wait();
        }
        value = newValue;
        full = true;
        notifyAll();
    }

    synchronized int take() throws InterruptedException {
        while (!full) {
            wait();
        }
        int result = value;
        full = false;
        notifyAll();
        return result;
    }
}

public class ProducerConsumerDemo {
    public static void main(String[] args) throws InterruptedException {
        Slot slot = new Slot();
        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) {
                    slot.put(i);
                }
            } catch (InterruptedException exception) {
                Thread.currentThread().interrupt();
            }
        }, "Producer");
        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("Consumed: " + slot.take());
                }
            } catch (InterruptedException exception) {
                Thread.currentThread().interrupt();
            }
        }, "Consumer");
        producer.start();
        consumer.start();
    }
}

```

```

        producer.join();
        consumer.join();
        System.out.println("Transfer complete");
    }
}

```

Output:

```

Consumed: 1
Consumed: 2
Consumed: 3
Transfer complete

```

Single slot overwriting और empty reading से predicate checks बचाती हैं। यह normal-run teaching example है; एक worker externally cancel हो तो partner को भी shutdown signal देना application की जिम्मेदारी है।

10.6 notify बनाम notifyAll

Notify एक unspecified waiting thread चुनती है। NotifyAll सभी waiters को eligible बनाती है; सभी एक साथ critical section में नहीं प्रवेश करतीं। Multiple condition predicates वाले monitor में notifyAll incorrect waiter selection से जुड़ी liveness problems घटा सकती है। यह fairness guarantee नहीं है।

10.7 Lost Notification

Condition flag के बिना केवल notification पर निर्भर code event miss कर सकता है। Wait शुरू होने से पहले notify हो जाए तो notification future wait के लिए store नहीं होती। इसलिए state predicate और lock-based protocol रखें।

10.8 Revision एवं अभ्यास

Wait object monitor चाहिए। Notify monitor तत्काल release नहीं करती। While predicate आवश्यक है। Wait wake होने पर monitor reacquire करती है। Interruption को silently swallow न करें।

अभ्यास: Producer पाँच numbers transfer करे। Buffer full/empty states की trace table बनाएँ।

MCQs: 1. wait किसकी method? A. Object B. Runnable C. Package D. System only 2. wait के लिए monitor ownership? A. Required B. नहीं C. priority 10 D. केवल main 3. Notify lock immediately छोड़ती? A. हाँ B. नहीं C. सभी locks D. केवल integer 4. Predicate check: A. while B. केवल if हमेशा C. कोई नहीं D. switch only

उत्तर: 1-A, 2-A, 3-B, 4-A.

UNIT III / CHAPTER 11

अध्याय 11: Thread Suspension, Resumption एवं Stopping

Legacy Methods और Cooperative Control

11.1 Syllabus और Version Context

इस अध्याय में `suspend()`, `resume()` और `stop()` के historical उद्देश्य समझेंगे। **Java 17 documentation में ये deprecated legacy controls हैं।** Newer JDKs में उनकी availability/behavior अलग हो सकती है। इसलिए runnable examples इन APIs पर निर्भर नहीं हैं।

11.2 Legacy Methods

Method	Historical उद्देश्य	समस्या
<code>suspend()</code>	Target execution रोकना	Held locks के साथ suspension; deadlock risk
<code>resume()</code>	Suspended target आगे चलाना	Unsafe suspension protocol पर निर्भर
<code>stop()</code>	Forceful termination	Protected state inconsistent हो सकती है

यह “sleep कराना” या “join करना” नहीं है। Sleep current thread की timed waiting और join another thread termination का wait है।

11.3 suspend का Deadlock Risk

यदि suspended thread monitor own करती हो, और resuming thread उसी monitor को पहले acquire करने की कोशिश करे, तो दोनों की progress रुक सकती है। Suspension monitor को सुरक्षित condition-wait protocol की तरह release नहीं करती।

11.4 stop का State Risk

Legacy asynchronous stopping thread को update के बीच रोक सकती है। Locks release होने पर अन्य threads partially updated object state देख सकती हैं। Graceful cancellation में worker safe points पर request पहचानकर cleanup करती है।

11.5 interrupt()

Interrupt cooperative request है; force kill नहीं। CPU loop को status check करना पड़ सकता है। Sleep/wait/join interruption पर `InterruptedException` देती हैं और status clear होता है। Handler cancellation complete करे या higher-level policy के लिए status restore करे।

11.6 Complete Cooperative Stop Program

```
import java.util.concurrent.CountDownLatch;

public class CooperativeStopDemo {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch started = new CountDownLatch(1);
        Thread worker = new Thread() -> {
            started.countDown();
            try {
                while (!Thread.currentThread().isInterrupted()) {
                    Thread.sleep(1000);
                }
            } catch (InterruptedException exception) {
                Thread.currentThread().interrupt();
            } finally {
                System.out.println("Worker cleanup completed");
            }
        };
        worker.start();
        started.await();
        worker.interrupt();
        worker.join();
        System.out.println("Worker stopped cooperatively");
    }
}
```

Output:

```
Worker cleanup completed
Worker stopped cooperatively
```

CountDownLatch startup coordination का library helper है। Main startup signal मिलने के बाद interrupt करती है। Worker interruption पहचानकर exit करती है।

11.7 Status Methods

Call	Status behavior
thread.isInterrupted()	Target status पढ़ती; clear नहीं
Thread.interrupted()	Current status पढ़कर clear
thread.interrupt()	Request/status set; blocked-operation behavior लागू

Catch के बाद बिना policy के खाली loop continue करना cancellation request खो सकता है।

11.8 Cooperative Pause/Resume Controller

```
class PauseController {
    private boolean paused;
    private boolean stopped;

    synchronized void pause() {
        paused = true;
    }
    synchronized void resumeWork() {
        paused = false;
        notifyAll();
    }
    synchronized void stopWork() {
        stopped = true;
        notifyAll();
    }
    synchronized boolean awaitPermission() throws InterruptedException {
        while (paused && !stopped) {
            wait();
        }
        return !stopped;
    }
}
```

Worker checkpoints पर awaitPermission() call करती है। False मिले तो exit करती है। Resume notification state update के साथ होती है। Stop paused worker को भी wake कर सकता है।

यह immediate freeze नहीं है: request worker के अगले checkpoint पर प्रभावी होती है। Worker अन्य interruptible blocking operation में हो, तो cancellation policy interrupt भी उपयोग कर सकती है।

11.9 Volatile Stop Flag

Simple CPU task में volatile boolean flag visibility दे सकता है। लेकिन flag अकेले indefinitely waiting thread को जगाता नहीं। Waiting worker के लिए notification या interruption की आवश्यकता हो सकती है।

11.10 Revision एवं अभ्यास

Deprecated controls केवल historical understanding के लिए हैं। Interrupt request है। Join completion wait है। Cooperative pause state predicate और wait-notify protocol से होती है।

अभ्यास: PauseController को counting worker में जोड़ें। Stop request paused worker को कैसे wake करती है, समझाएँ।

MCQs: 1. Interrupt: A. Force kill हमेशा B. Cooperative request C. Constructor D. Priority setting 2. isInterrupted status clear? A. हाँ B. नहीं C. केवल main D. हमेशा exception 3. Legacy suspend risk: A. Deadlock B. केवल compile speed C. Array size D. String pool

उत्तर: 1-B, 2-B, 3-A.

UNIT III / CHAPTER 12

अध्याय 12: Revision, Question Bank एवं Mini Project

Unit III का समेकित अभ्यास

12.1 Revision Table

Topic	मुख्य अवधारणा
Throwable	Thrown conditions का root
Checked exceptions	Catch-or-declare obligation
Unchecked exceptions	RuntimeException/Error branches
try-catch	Matching failure handling
Multi-catch	Unrelated alternatives, one handler
throw / throws	Actual throw / declaration
finally	Cleanup, termination caveats
Custom exceptions	Domain-specific failure types
Thread / Runnable	Execution control / task contract
start / run	New execution / ordinary method call
join	Termination wait
Thread.State	Six enum states
synchronized	Monitor mutual exclusion और visibility
wait / notifyAll	Condition coordination
interrupt	Cooperative cancellation signal

12.2 Frequently Confused Concepts

Pair	अंतर
Checked / compile-time occurrence	Checked obligation compile time; exception runtime
throw / throws	Object expression / type list

Pair	अंतर
Multiple catches / multi-catch	Separate blocks / alternatives in one block
final / finally	Modifier / cleanup block
start / run	New thread schedule / current-thread call
wait / sleep	Monitor condition release / timed delay holding monitors
notify / monitor release	Notification lock ownership तुरंत नहीं देती
volatile / atomic increment	Visibility / compound operation protection
interrupt / stop	Cooperative signal / unsafe legacy termination
join / failure rethrow	Termination wait / exception reporting अलग

12.3 Solved Output Questions

Q1: Exception flow

```
int divisor = 0;
try {
    System.out.print("A ");
    System.out.println(10 / divisor);
    System.out.print("B ");
} catch (ArithmeticException exception) {
    System.out.print("C ");
} finally {
    System.out.print("D ");
}
```

उत्तर: A C D I B skip होती है।

Q2: No failure path

```
try {
    System.out.print("Try ");
} finally {
    System.out.print("Finally ");
}
```

उत्तर: Try Finally। Catch आवश्यक नहीं जब finally मौजूद है।

Q3: Type classification NumberFormatException? **Unchecked**। IOException? **Checked**। StackOverflowError? **Unchecked Error branch**।

Q4: Direct run Main से Thread object's run() directly call करने पर नया thread बनता है? **नहीं**। Body calling thread में चलती है।

Q5: Completion Worker start, फिर join, फिर print("Done")। Done कब? **Target worker termination के बाद**, यदि join normally complete करे।

Q6: Synchronized count दो threads same Counter में प्रत्येक 500 increments करती हैं और main दोनों को join करती है। Expected count? **1000**, यदि increments same lock से correctly protected हैं।

Q7: Waiting without monitor

```
Object lock = new Object();
// lock.wait(); // Must own lock and handle checked interruption
```

बिना monitor ownership actual wait call IllegalMonitorStateException दे सकती है।

InterruptedException obligation भी compile time पर account करनी होगी।

12.4 MCQ Question Bank

1. Throwable का subclass branch? A. Exception B. String C. Runnable D. Scanner
2. Integer division by zero? A. ClassNotFoundException B. ArithmeticException C. IOException D. InterruptedException
3. Checked example? A. NullPointerException B. ArithmeticException C. IOException D. NumberFormatException
4. RuntimeException category? A. Checked B. Interface C. Thread state D. Unchecked
5. Error branch category? A. Unchecked B. Checked always C. Primitive D. Runnable
6. Catch order? A. Alphabetical B. Specific before broad C. Broad always first D. कोई rule नहीं
7. Multi-catch alternatives related हो सकती हैं? A. केवल RuntimeException B. केवल Error C. नहीं D. हमेशा
8. Actual object throw keyword? A. throws B. finally C. final D. throw
9. Exception type declaration? A. throws B. throw C. new only D. notify
10. Finally always despite process kill? A. केवल checked B. नहीं C. हाँ D. केवल JVM पर हाँ
11. Custom checked base? A. Thread B. String C. Exception D. RuntimeException
12. Custom unchecked base? A. IOException B. Runnable C. Scanner D. RuntimeException
13. Exception message? A. getMessage() B. getName() C. getPriority() D. join()
14. Thread task interface? A. Package B. Runnable C. Throwable D. String
15. New thread execution? A. getState B. sleep C. start() D. run directly
16. Direct run executes? A. हमेशा नया thread B. केवल daemon C. JVM shutdown D. Calling thread
17. Restart same terminated Thread? A. नहीं B. हाँ C. केवल interrupt D. केवल priority 10
18. Target termination wait? A. getName B. join C. notify D. suspend
19. Current thread reference? A. mainOnly B. getCause C. currentThread D. newCurrent
20. Thread.State enum में कौन नहीं? A. NEW B. RUNNABLE C. TERMINATED D. RUNNING
21. Sleep का usual state? A. TIMED_WAITING B. NEW C. TERMINATED D. Always BLOCKED
22. Monitor acquire wait state? A. DEFAULT B. BLOCKED C. NEW D. TERMINATED
23. Platform normal priority constant? A. 10 B. 0 C. 5 D. 1
24. High priority first finish guarantee? A. हाँ B. केवल Java C. केवल int D. नहीं
25. Sleep monitor release? A. नहीं B. सभी C. केवल this D. हमेशा एक

26. Instance synchronized lock? A. package B. this C. Class हमेशा D. Thread name
27. Static synchronized lock? A. String literal हमेशा B. null C. Declaring Class object D. this
28. Volatile counter++ atomic? A. हाँ B. केवल long C. केवल int D. नहीं
29. Object wait requires? A. Monitor ownership B. priority 10 C. final class D. daemon
30. wait method declared in? A. Package B. Object C. Runnable D. System
31. notify monitor तुरंत छोड़ती? A. सभी monitors B. केवल Main C. नहीं D. हाँ
32. Condition checking best form? A. if हमेशा पर्याप्त B. switch only C. no check D. while
33. notifyAll wakes? A. All wait-set threads to compete B. सभी एक साथ lock own C. नई threads बनाती D. सभी terminate
34. Spurious wakeups? A. only priority 10 B. सम्भव C. असम्भव D. compile error
35. interrupt force kill? A. only daemon B. only Runnable C. नहीं D. हमेशा
36. isInterrupted clears status? A. हाँ B. हमेशा exception C. constructor D. नहीं
37. Thread.interrupted current status clear? A. हाँ B. नहीं C. केवल child D. only priority 5
38. Legacy suspend main risk? A. String interning B. Deadlock C. Array overflow D. File extension
39. Legacy stop risk? A. Same as join B. Always safe C. Inconsistent shared state D. केवल slow compile
40. Join worker exception automatically rethrows? A. हमेशा B. केवल checked C. केवल RuntimeException D. नहीं

12.5 Answer Key

प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर
1	A	11	C	21	A	31	C
2	B	12	D	22	B	32	D
3	C	13	A	23	C	33	A
4	D	14	B	24	D	34	B
5	A	15	C	25	A	35	C
6	B	16	D	26	B	36	D
7	C	17	A	27	C	37	A
8	D	18	B	28	D	38	B
9	A	19	C	29	A	39	C
10	B	20	D	30	B	40	D

12.6 Short-Answer Questions

1. Exception और java.lang.Error का अंतर बताइए।

2. Checked exception की compile-time obligation क्या है?
3. Throw और throws में अंतर लिखिए।
4. Finally block कब execute हो सकती है और कब नहीं?
5. Thread और process का अंतर लिखिए।
6. Start और run का अंतर समझाइए।
7. Thread.State की छह values लिखिए।
8. Sleep और wait का अंतर बताइए।
9. Volatile increment atomic क्यों नहीं है?
10. Interrupt और forceful stop में अंतर लिखिए।

12.7 Long-Answer Questions

1. Exception hierarchy का diagram और classification समझाइए।
2. Multiple catches, nested try और finally का flow समझाइए।
3. Custom exception class example सहित बनाइए।
4. Thread तथा Runnable से creation के programs लिखिए।
5. Thread lifecycle और priorities समझाइए।
6. Race condition और synchronized समाधान explain कीजिए।
7. Producer-consumer problem में wait-notifyAll protocol समझाइए।
8. Legacy stopping/suspension के risks और cooperative alternatives बताइए।

12.8 Mini Project: Concurrent Marks Processing

दो workers marks process करती हैं। Invalid marks के लिए custom checked exception है। Shared summary synchronized है। Workers का internal execution order बदल सकता है; final summary joins के बाद stable है।

Save as MarksProcessingApp.java:

```
class InvalidMarkException extends Exception {
    private static final long serialVersionUID = 1L;
    InvalidMarkException(int mark) {
        super("Invalid mark: " + mark);
    }
}

class Summary {
    private int accepted;
    private int rejected;
    private int total;

    synchronized void accept(int mark) {
        accepted++;
        total += mark;
    }
    synchronized void reject() {
        rejected++;
    }
    synchronized void display() {
        System.out.println("Accepted: " + accepted);
        System.out.println("Rejected: " + rejected);
    }
}
```

```

        System.out.println("Total valid marks: " + total);
    }
}

class MarksTask implements Runnable {
    private final int[] marks;
    private final Summary summary;

    MarksTask(int[] marks, Summary summary) {
        this.marks = marks.clone();
        this.summary = summary;
    }

    static void validate(int mark) throws InvalidMarkException {
        if (mark < 0 || mark > 100) {
            throw new InvalidMarkException(mark);
        }
    }

    @Override
    public void run() {
        for (int mark : marks) {
            if (Thread.currentThread().isInterrupted()) {
                return;
            }
            try {
                validate(mark);
                summary.accept(mark);
            } catch (InvalidMarkException exception) {
                summary.reject();
            }
        }
    }
}

public class MarksProcessingApp {
    public static void main(String[] args) throws InterruptedException {
        Summary summary = new Summary();
        Thread first = new Thread(
            new MarksTask(new int[]{70, 80, 120}, summary));
        Thread second = new Thread(
            new MarksTask(new int[]{90, -5, 60}, summary));
        first.start();
        second.start();
        first.join();
        second.join();
        summary.display();
    }
}

```

12.9 Commands एवं Output

```

javac MarksProcessingApp.java
java MarksProcessingApp

```

```

Accepted: 4
Rejected: 2
Total valid marks: 300

```

Valid values 70,80,90,60 हैं। Invalid 120 और -5 हैं। Accepted total $70+80+90+60 = 300$ । Inputs इस example में non-null हैं। Full application में input validation और interruption पर main cleanup policy भी जोड़ें।

12.10 Project Extensions

- Rejected messages अलग synchronized log में store करें।
- Worker tasks को packages में रखें।
- Producer-consumer queue से marks supply करें।
- Cancellation request पर दोनों workers interrupt और join करें।
- Average valid marks निकालें; zero accepted case handle करें।
- Console output में processing order को fixed मानने के स्थान पर counts verify करें।

12.11 Official Reference Reading

यह पुस्तक Java 17-compatible teaching examples उपयोग करती है। API details के लिए official documentation पढ़ें:

- [Java SE 17 Thread API](#)
- [Legacy thread-control explanation](#)
- [Java Language Specification, Chapter 11 \(Exceptions\)](#)
- [Java SE 17 Object API](#)

12.12 अंतिम Revision Checklist

क्या आप try में failure के बाद skipped statements पहचान सकते हैं? क्या catch order सही लिख सकते हैं? क्या checked obligation runtime occurrence से अलग है? क्या start/run और wait/sleep स्पष्ट हैं? क्या shared updates same lock से protected हैं? क्या interrupted waiting को सही policy से handle करते हैं? क्या पुराने stop/suspend के बजाय safe points का उपयोग कर सकते हैं? इन questions के छोटे examples स्वयं चलाएँ।