



A T M E
College of Engineering

GOLD

QS I-GAUGE
INDIAN CYBER RATING



Object-Oriented Programming with Java

Module 4: Exception handling and fundamentals



Exceptions

Exception-Handling Fundamentals

Definition: An exception is an event that occurs during program execution that disrupts the normal flow of instructions.

Examples:

- Division by zero
- Array index out of bounds
- Null object reference
- File not found

Java handles exceptions using try, catch, throw, throws, and finally.

Basic keywords

Java provides five important keywords:

Keyword

try

catch

throw

throws

finally

Purpose

Contains code that may cause an exception

Handles the exception

Explicitly throws an exception

Declares exceptions that a method may throw

Contains code that normally executes whether exception occurs or not

Exception Types

Java exceptions are mainly divided into:

A. Checked Exceptions

These are checked by the compiler at compile time.

Examples:

- IOException
- SQLException
- ClassNotFoundException
- FileNotFoundException

```
FileReader f = new FileReader("abc.txt");
```

Unchecked Exceptions

These occur during runtime.

They are generally subclasses of RuntimeException.

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException

```
int x = 10 / 0;
```

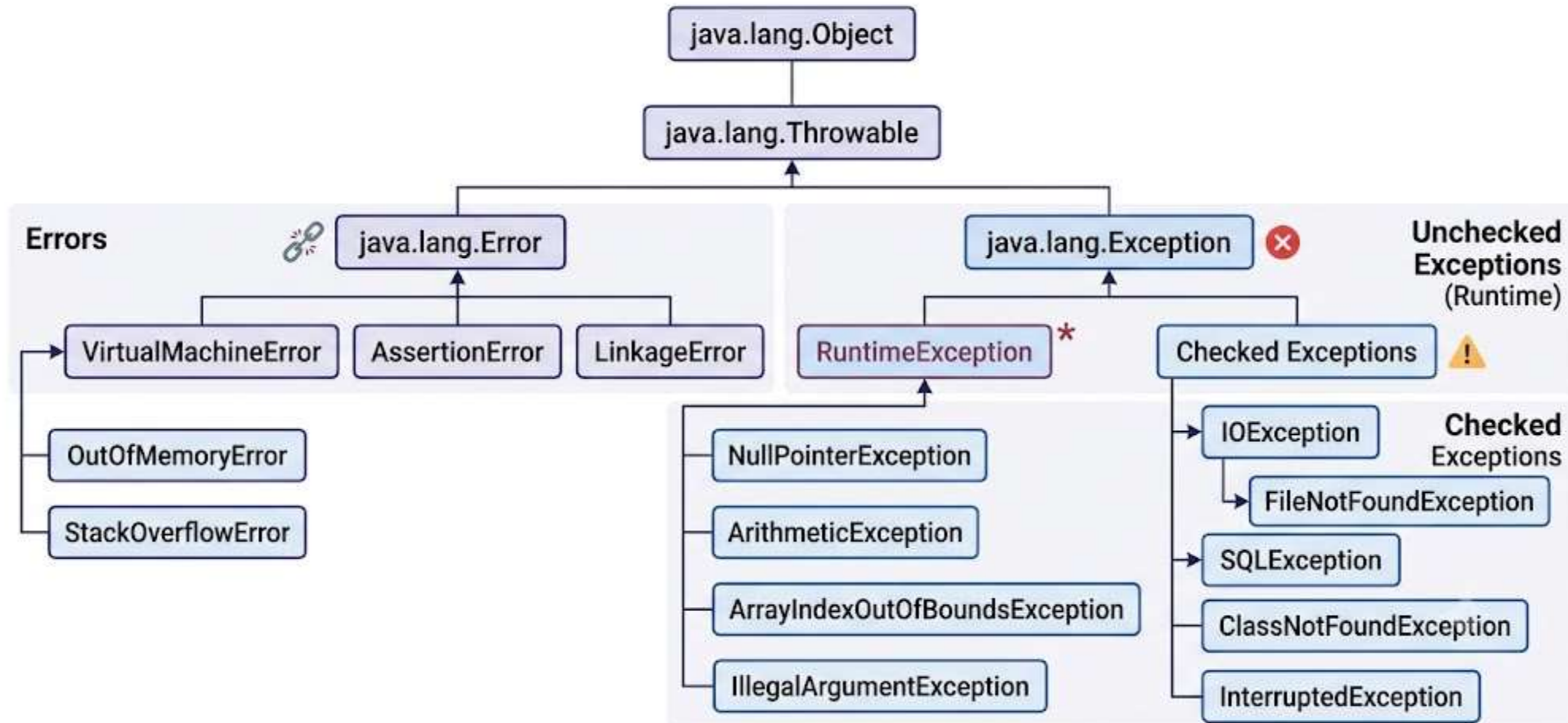
Errors

Errors represent serious problems that applications generally should not try to handle.

Examples:

- OutOfMemoryError
- StackOverflowError

Java Exception Hierarchy



Uncaught Exceptions

An exception that is not handled by the program is called an **uncaught exception**.

Example:

```
class Demo {  
    public static void main(String[] args) {  
        int a = 10 / 0;  
        System.out.println("Hello");  
    }  
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

Using try and catch

The try block contains statements that may generate an exception.
The catch block handles the exception.

```
try {  
    // risky code  
}  
catch(ExceptionType e) {  
    // handling code  
}
```

```
class Demo {  
    public static void main(String[] args) {  
        try {  
            int a = 10 / 0;  
        }  
        catch (ArithmeticException e) {  
            System.out.println("Cannot divide by zero");  
        }  
    }  
}
```

Multiple Catch Clauses

A single try block can have multiple catch blocks.

Example

```
try {  
    int a = 10 / 0;  
}  
catch (ArithmeticException e) {  
    System.out.println("Arithmetic error");  
}  
catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Array error");  
}  
catch (Exception e) {  
    System.out.println("Some other exception");  
}
```

Nested try Statements

A try block can be placed inside another try block.

Example

```
try {  
    System.out.println("Outer try");  
  
    try {  
        int x = 10 / 0;  
    }  
    catch (ArithmeticException e) {  
        System.out.println("Inner catch");  
    }  
}  
catch (Exception e) {  
    System.out.println("Outer catch");  
}
```

throw Keyword

The throw keyword is used to **explicitly throw an exception**.

Syntax

```
throw new ExceptionType();
```

```
class Demo {  
    public static void main(String[] args) {  
  
        int age = 15;  
  
        if (age < 18) {  
            throw new ArithmeticException("Not eligible");  
        }  
    }  
}
```

throws Keyword

The throws keyword is used in a method declaration to indicate that the method **may throw one or more exceptions**.

Syntax

```
returnType methodName() throws  
ExceptionType {  
}
```

Example

```
void readFile() throws  
IOException {  
    // file handling code  
}
```

finally Block

The finally block normally executes whether an exception occurs or not.

Example

```
try {  
    int x = 10 / 2;  
}  
catch (ArithmeticException e) {  
    System.out.println("Error");  
}  
finally {  
    System.out.println("Finally block executed");  
}
```

Chained Exceptions

Chained exceptions allow one exception to be associated with another exception.

It is useful when one exception is caused by another exception.

```
class Demo {
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            Exception e1 =
```

```
                new Exception("Original exception");
```

```
            Exception e2 =
```

```
                new Exception("New exception", e1);
```

```
            throw e2;
```

```
        }
```

```
        catch (Exception e) {
```

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

```
            System.out.println("Cause: " + e.getCause());
```

```
        }
```

```
    }
```

```
}
```

MULTITHREADED PROGRAMMING

14. What is Multithreading?

Multithreading is the process of executing multiple threads concurrently within a program.

A **thread** is a lightweight unit of execution.

The Main Thread

The **main thread** is the thread that starts when a Java application begins execution

```
class Demo {  
    public static void main(String[] args) {  
  
        Thread t = Thread.currentThread();  
  
        System.out.println("Current thread: " + t);  
    }  
}
```

Creating a Thread Using Runnable

Another common approach is implementing the Runnable interface.

```
class MyThread implements Runnable {  
  
    public void run() {  
        System.out.println("Thread is running");  
    }  
}  
  
class Demo {  
    public static void main(String[] args) {  
  
        MyThread obj = new MyThread();  
  
        Thread t = new Thread(obj);  
  
        t.start();  
    }  
}
```

isAlive() Method

The isAlive() method checks whether a thread is still running.

Syntax

threadName.isAlive()

```
class MyThread extends Thread {  
  
    public void run() {  
        System.out.println("Thread running");  
    }  
}
```

```
class Demo {  
    public static void main(String[] args) {  
  
        MyThread t = new MyThread();  
  
        System.out.println(t.isAlive());  
  
        t.start();  
  
        System.out.println(t.isAlive());  
    }  
}
```

join() Method

The join() method makes one thread **wait until another thread finishes.**

Example

```
class MyThread extends Thread {
```

```
    public void run() {  
        for(int i = 1; i <= 3; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
class Demo {
```

```
    public static void main(String[] args)  
        throws InterruptedException {
```

```
        MyThread t = new MyThread();
```

```
        t.start();
```

```
        t.join();
```

```
        System.out.println("Main thread continues");
```

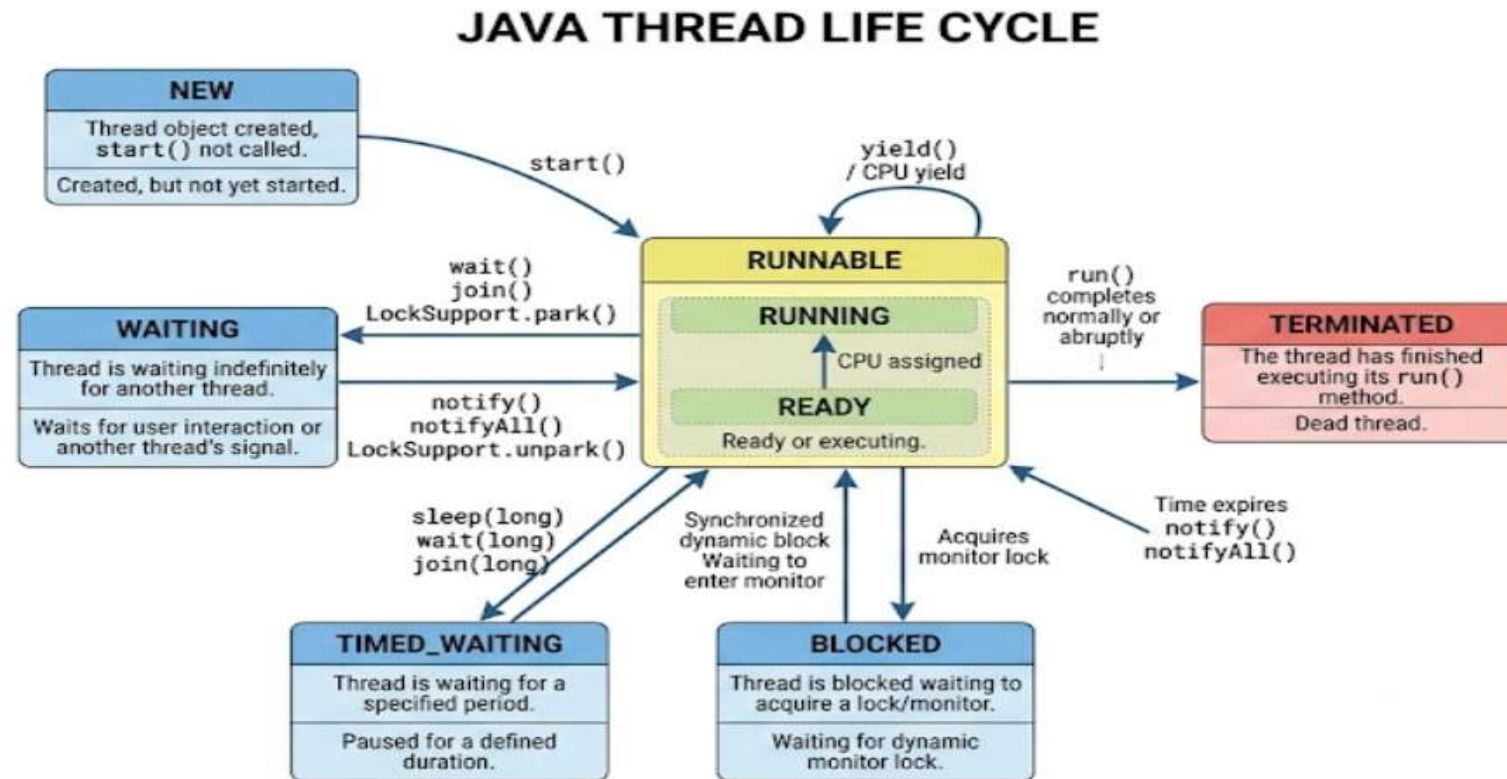
```
    }
```

Thread States in Java

A **thread state** represents the current condition of a thread during its life cycle.

Java provides these states through the Thread.State enumeration:

Thread Life Cycle



Thread Priorities in Java

Thread priority tells the thread scheduler the relative importance of a thread.

Java provides priorities from **1 to 10**.

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

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running");  
    }  
}
```

```
class Demo {  
    public static void main(String[] args) {  
  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
  
        t1.setPriority(Thread.MAX_PRIORITY);  
        t2.setPriority(Thread.MIN_PRIORITY);  
  
        t1.start();  
        t2.start();  
    }  
}
```



A T M E
College of Engineering

GOLD

QS I-GAUGE
INDEPENDENT RATING



THANK YOU



ATME

College of Engineering

GOLD

QS

I-GAUGE

INDIAN CPEI 3 STAR RATING

