Exception Handling & Multithreaded Programming.

Prepared By:
D R GANDHI

Course Outcome

- Implement exception handling and multithreading in object oriented programs.

Learning Outcomes

- 4.1.1 Fundamentals of Exception and Errors, Types of Exception.
- 4.1.2 Using try and catch in Exception, Multiple catch clauses, Use of nested try statements.
- 4.1.3 Throw and throws keywords, and finally clause.
- 4.1.4 Built in exceptions, creating own exception subclasses, Java Optional class.
- 4.2.1 Basics of Multithreading, The Java thread model and main thread, Creation of thread by extending Thread class, implementing Runnable interface.
- 4.2.2 Life cycle of a thread.
- 4.2.3 Thread priorities, Thread synchronization, inter thread communication, alive () &join () in thread.
- 4.2.4 Exception handling in threads

Exception Handling-Fundamentals

- An exception is an abnormal condition that arises in a code sequence at run time.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:
  - provides syntactic mechanisms to signal, detect and handle errors.
  - ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors.
  - brings run-time error management into object-oriented programming.

Exception Handling-Fundamentals

- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, therefore these exceptions are needs to be handled.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- An exception can be caught to handle it or pass it on. In any way exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exception Handling-Fundamentals

- An exception can occur for many different reasons, some of them are as given below:
- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.
- Exceptions are caused by users, programmers or when some physical resources get failed.
- The Exception Handling in java is one of the powerful mechanisms to handle the exception (runtime errors), so that normal flow of the application can be maintained.

Exception Handling-Fundamentals

- In Java there are three categories of Exceptions:
- **Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. Example, IOException, SQLException etc.
- **Runtime exceptions:** An Unchecked exception is an exception that occurs during the execution, these are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.
- Example : ArithmeticException, NullPointerException, Array Index out of Bound exception.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Example: OutOfMemoryError, VirtualMachineErrorException.

Exception Handler

- ✓ It is a block of code to handle thrown exception.
- ✓ If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

Types of Errors

- There are several types of errors that occur in Java.
- syntax errors.
- runtime errors.
- logical errors.

Types of Errors

Syntax Errors or Compilation Errors:

- ✓ These occur when the code violates the rules of the Java syntax.
- ✓ These errors are usually caught by the Java compiler during the compilation phase.

Example 1: Missing semicolon

Code:

```
public class Main
{
  public static void main(String[] args)
  {
    System.out.println("Hello, World!")
  }
}
```

Types of Errors

Runtime Errors:

- ✓ These errors occur when the code encounters an unexpected behavior during its execution.
- ✓ These errors are usually caused by flawed logic or incorrect assumptions in the code and can be difficult to identify and fix.
- ✓ The most common runtime errors are as follows:
- ✓ 1. Dividing an integer by zero.
- ✓ 2. Accessing an element that is out of range of the array.
- ✓ 3. Trying to store a value into an array that is not compatible type.

Types of Errors

Runtime Errors:

```
public class DivisionByZeroError
{
public static void main(String[] args)
{
 int a = 20, b = 5, c = 5;
 int z = a/(b-c); // Division by zero.

 System.out.println("Result: " +z);
  }
}
```

```
Output:
      Exception in thread "main" java.lang.ArithmeticException: / by zero
      at errorsProgram.DivisionByZeroError.main(DivisionByZeroError.java:8)
```

Types of Errors

Logical Errors:

- ✓ These occur when the program is executing correctly, but the result is not what was intended.
- ✓ These errors can be difficult to identify and fix, as the program is running correctly but producing unintended results.
- ✓ Logical errors are not detected either by Java compiler or JVM (Java runtime system).
- ✓ The programmer is entirely responsible for them. They can be detected by application testers when they compare the actual result with its expected result.

Types of Errors

Logical Errors:

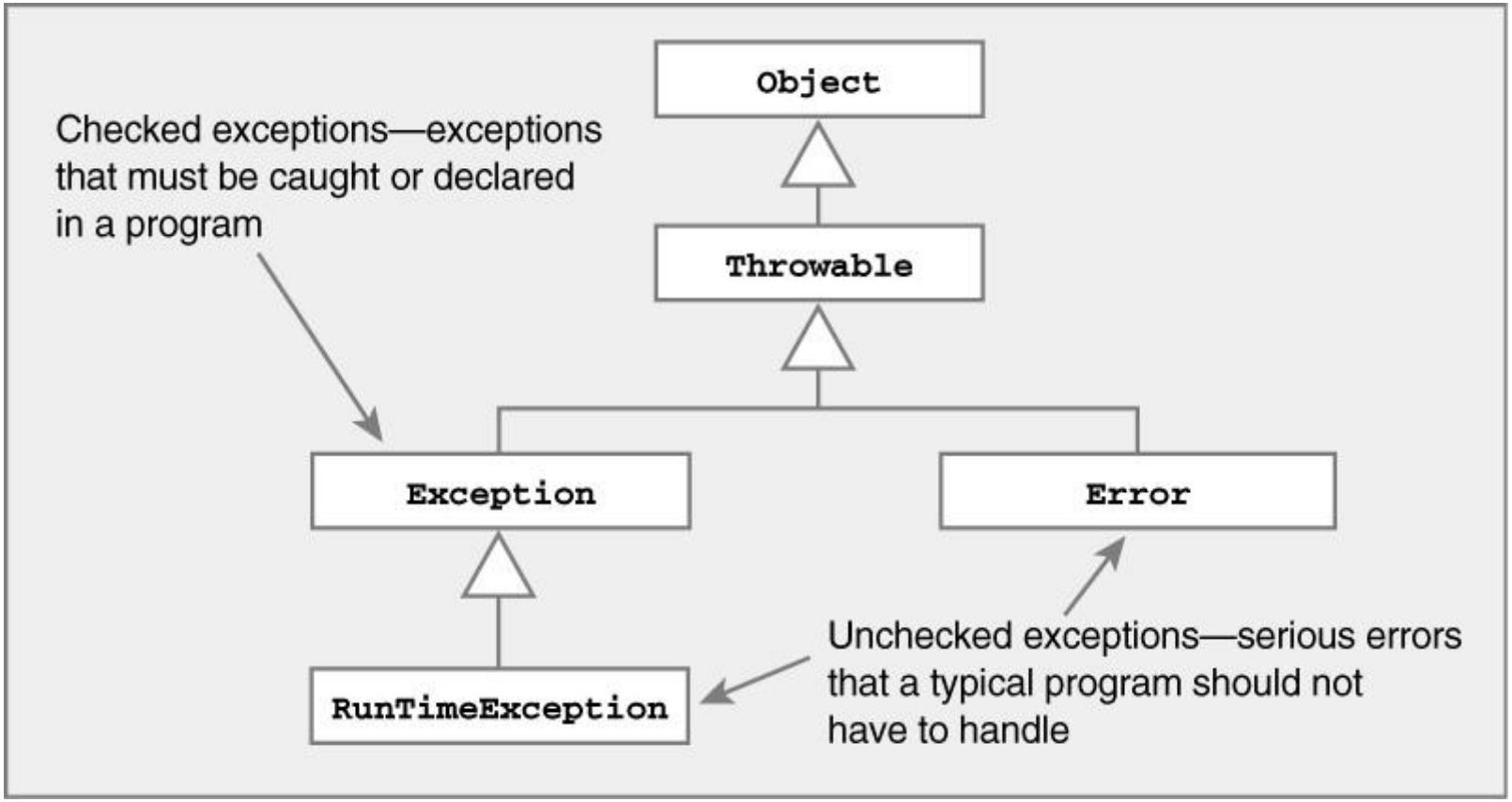
```
public class LogicalErrorEx
{
public static void main(String[] args)
{
 int a[]={1, 2 , 5, 6, 3, 10, 12, 13, 14};

System.out.println("Even Numbers:");
for(int i = 0; i <a.length; i++)
{
if(a[i] / 2 == 0) // Using wrong operator.
{
  System.out.println(a[i]);
 }
}
 }
}
```

```
Output:
        Even Numbers:
        1
```

Exception Handling

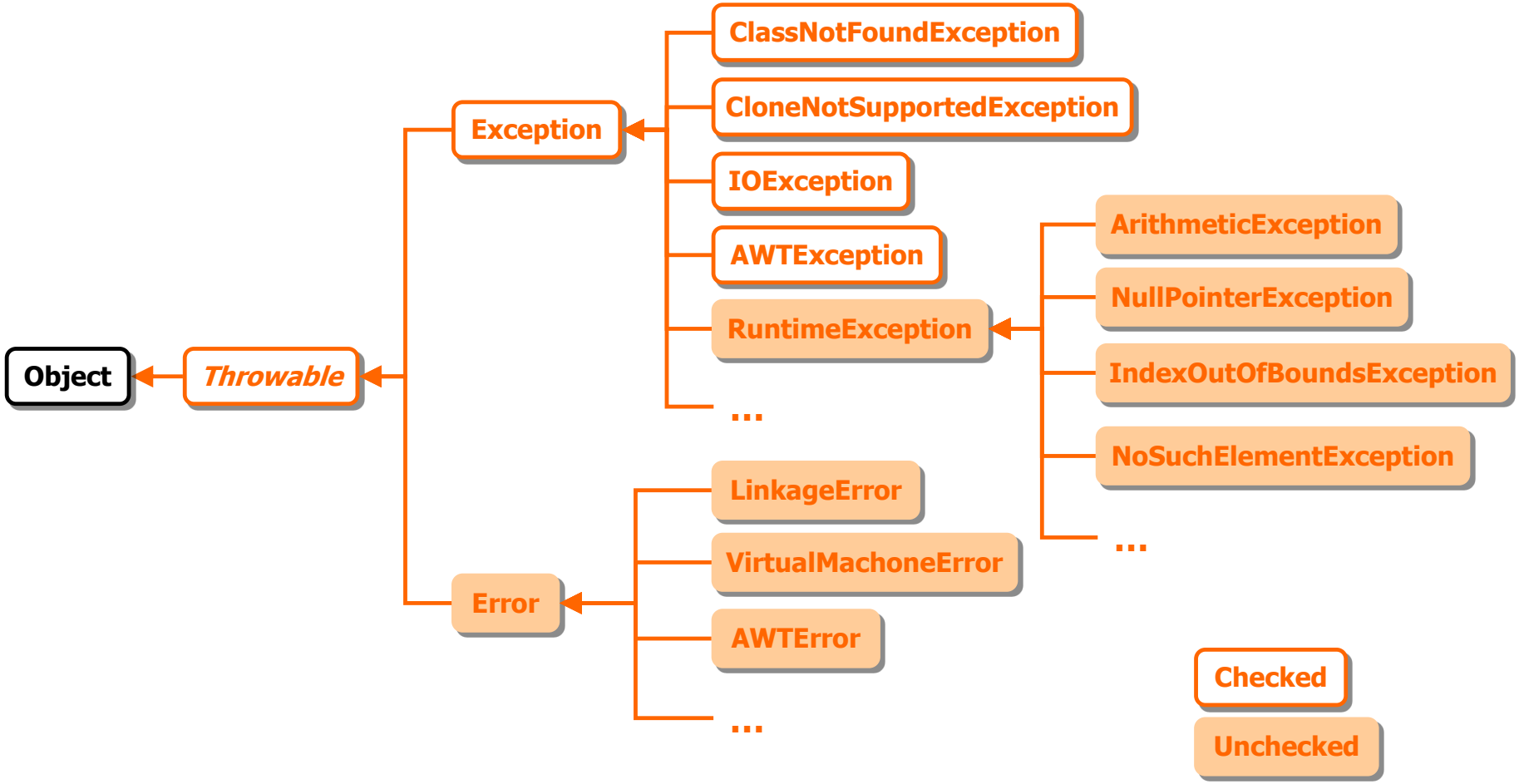
- ✓ All exception classes are subtypes of the java.lang.Exception class.
- ✓ The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Representing Exceptions



Representing Exceptions

- Java Exception class hierarchy



Exception and Error class

- Exceptions and errors both are subclasses of Throwable class.
- **Exceptions** are the problems which can occur at runtime and compile time.
- It mainly occurs in the code written by the developers. Exceptions are divided into two categories such as checked exceptions and unchecked exceptions.
- The **error** indicates a problem that mainly occurs due to the lack of system resources and our application should not catch these types of problems.
- Some of the examples of errors are system crash error and out of memory error.
- Errors mostly occur at runtime that's they belong to an unchecked type.

Exception and Error class

Basis of Comparison	Exception	Error
Recoverable/ Irrecoverable	Exception can be recovered by using the try-catch block.	An error cannot be recovered.
Type	It can be classified into two categories i.e. checked and unchecked.	All errors in Java are unchecked.
Occurrence	It occurs at compile time or run time.	It occurs at run time.
Package	It belongs to java.lang.Exception package.	It belongs to java.lang.Error package.
Known or unknown	Only checked exceptions are known to the compiler.	Errors will not be known to the compiler.
Causes	It is mainly caused by the application itself.	It is mostly caused by the environment in which the application is running.
Example	**Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundException, NullPointerException, ArithmaticException	Java.lang.StackOverFlow, java.lang.OutOfMemoryError

Checked vs. Unchecked Exceptions

- Subclasses of error and RuntimeException are known as *unchecked exceptions.*
- These exceptions are not checked by the compiler
- These need not be caught or declared to be thrown in your program.
- These are generally programming logical errors that shall be been fixed in compiled-time, rather than leaving it to runtime exception handling.
- All the other exception are called *checked exceptions*.
- They are checked by the compiler and must be caught or declared to be thrown.

Uncaught Exceptions

```
class Exc0 {
    public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
    }
}
```

- JRE throws exception and exc0 stops.
- This exception must be caught by exception handler.
- We have not supplied exception handler so it is caught by default exception handler provided by JRE.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Exception Handling in Java

- Java exception handling is managed by via five keywords: **try, catch, throw, throws,** and **finally.**
  - Program statements to monitor are contained within a **try** block.
  - If an exception occurs within the **try** block, it is thrown.
  - Code within **catch** block catch the exception and handle it.
  - **throw** is used to manually throw an exception.
  - **throws** keyword is used to declare that a method may throw one or some exceptions.
  - Code that must be executed before a method returns is put in a **finally** block.

Exception Handling in Java

- Java's exception handling consists of three operations:
  - Declaring exceptions;
  - Throwing an exception; and
  - Catching an exception.
- **Declaring Exceptions**
  - A Java method must declare in its signature the types of checked exception it may "throw" from its body, via the keyword "throws".
  - For example,

```
public void methodD() throws XxxException, YyyException {
      // method body throw XxxException and YyyException
}
```

Exception Handling in Java

- **Throwing an Exception**
  - When a Java operation encounters an abnormal situation, the method containing the erroneous statement shall create an appropriate Exception object and throw it to the Java runtime via the statement "throw XxxException".

```
public void methodD() throws XxxException, YyyException {// method's signature
    // method's body ... ...
    // XxxException occurs
    if ( ... )
     throw new XxxException(...);//construct an XxxException object and throw to JVM
    ...
     // YyyException occurs
    if ( ... ) throw new YyyException(...);//construct an YyyException object
and throw to JVM
     ..
}
```

Exception Handling in Java

- **Catching an Exception**
  - When a method throws an exception, the JVM searches backward through the call stack for a matching exception handler.
  - Each exception handler can handle one particular class of exception.
  - For example,

```
public void methodC() { // no exception declared
  ......
  try{
    ......
    // uses methodD() which declares XxxException & YyyException
    methodD();
    ......
   } catch (XxxException ex) {
        // Exception handler for XxxException
        ......
   } catch (YyyException ex} {
     // Exception handler for YyyException
        ......
   } finally { // optional
        // These codes always run, used for cleaning up
         ......
   }
   ......
}
```

Exception-handling block

- General framework for an exception-handling:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

- ExceptionType is the type of exception that has occurred.

try and catch statement

try block :

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.

catch block:

- Java catch block is used to handle the Exception.
- It must be used after the try block only.
- The catch block that follows the try is checked, if the type of exception that occurred is listed in the catch block then the exception is handed over to the catch block that handles it.
- You can use multiple catch block with a single try.

try and catch statement

- To handle a run-time error/ exception, simply enclose the code that you want to monitor inside a try block.
- Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.
- Following program includes a try block and a catch clause which processes the ArithmeticException generated by the division-by-zero error:

Example

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output:

Division by zero.

After catch statement.

try and catch statement

- A try and its catch statement form a unit.
- The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.
- The statements that are protected by the **try** must be surrounded by curly braces.

Multiple Catch Clauses

- If more than one exception can occur, then we use multiple catch clauses.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed and execution continues after the try/catch block.

try and catch statement

Example:

```
class demoTry
{
        public static void main(String[] args)
        {
                try
                {
                        int arr[]={1,2,3};
                        arr[3]=3/0;
                }
                catch(ArithmeticException ae)
                {
                        System.out.println("Divide by zero :: " + ae);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array index out of bound exception :: "+e);
                }
        }
}
```

Output:

```
divide by zero :: java.lang.ArithmeticException: / by zero
```

Example

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Caution

- Exception subclass must come before any of of their superclasses.
- A **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass.
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error

Example

```
/*  This program contains an error.

    A subclass must come before its superclass in
    a series of catch statements. If not,
    unreachable code will be created and a
    compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

- A **try** statement can be inside the block of another try.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the JRE will handle the exception.
- If a method call within a try block has try block within it, then then it is still nested try.

Nested try Statements

- Syntax:

```
Syntax:
try
{
        Statement 1;
        try
        {
        //Protected code
        }
        catch(ExceptionName e1)
        {
        //Catch block1
        }
}
catch(ExceptionName1 e2)
{
//Catch block 2
}
```

Nested try Statements

Example:

```
class demoTry1
{
        public static void main(String[] args)
        {
                try
                {
                        int arr[]={5,0,1,2};
                        try
                        {
                                arr[4] =arr[3]/arr[1];
                        }
                        catch(ArithmeticException  e)
                        {
                                System.out.println("divide by zero :: "+e);
                        }
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("array index out of bound exception :: "+e);
                }
                catch(Exception e)
                {
                        System.out.println("Generic exception :: "+e);
                }
                System.out.println("Out of try..catch block");
        }
}
```

Output:

```
divide by zero :: java.lang.ArithmeticException: / by zero
Out of try..catch block
```

Example

```
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

throw

- The throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception using throw keyword.
- Only object of Throwable class or its sub classes can be thrown.
- Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

throw

- It is possible for your program to throw an exception explicitly.

  throw *ThrowableInstance*

- *ThrowableInstance* must be an object of type **Throwable** or a subclass **Throwable.**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator

Example -throw unchecked exception

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

Example -throw Statements

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // re-throw the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

Output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

throws

- The throws keyword is used to declare an exception.
- If a method does not handle a checked exception, the method must declare it using the throws keyword.
- The throws keyword appears at the end of a method's signature.
- You can declare multiple exceptions.

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- throws clause lists the types of exceptions that a method might throw.
  -
```
type method-name(parameter-list) throws exception-list
    {
        // body of method
    }
```
- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

Example: incorrect program

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

Example: corrected version

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

Inside throwOne.

Caught java.lang.IllegalAccessException: demo

Example2

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    throw new IOException("File not found");
  }

  public static void main(String[] args) {
    try {
      findFile();
      System.out.println("Rest of code in try block");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output:

File not found

Example

- The findFile() method throws an IOException with the message we passed to its constructor. We are specifying it in the throws clause because it is the checked exception.
- The methods that call this findFile() method need to either handle this exception or specify it using the throws keyword themselves.
- We have handled this exception in the main() method.
- The flow of program execution transfers from the try block to the catch block when an exception is thrown. So, the rest of the code in the try block is skipped and statements in the catch block are executed.

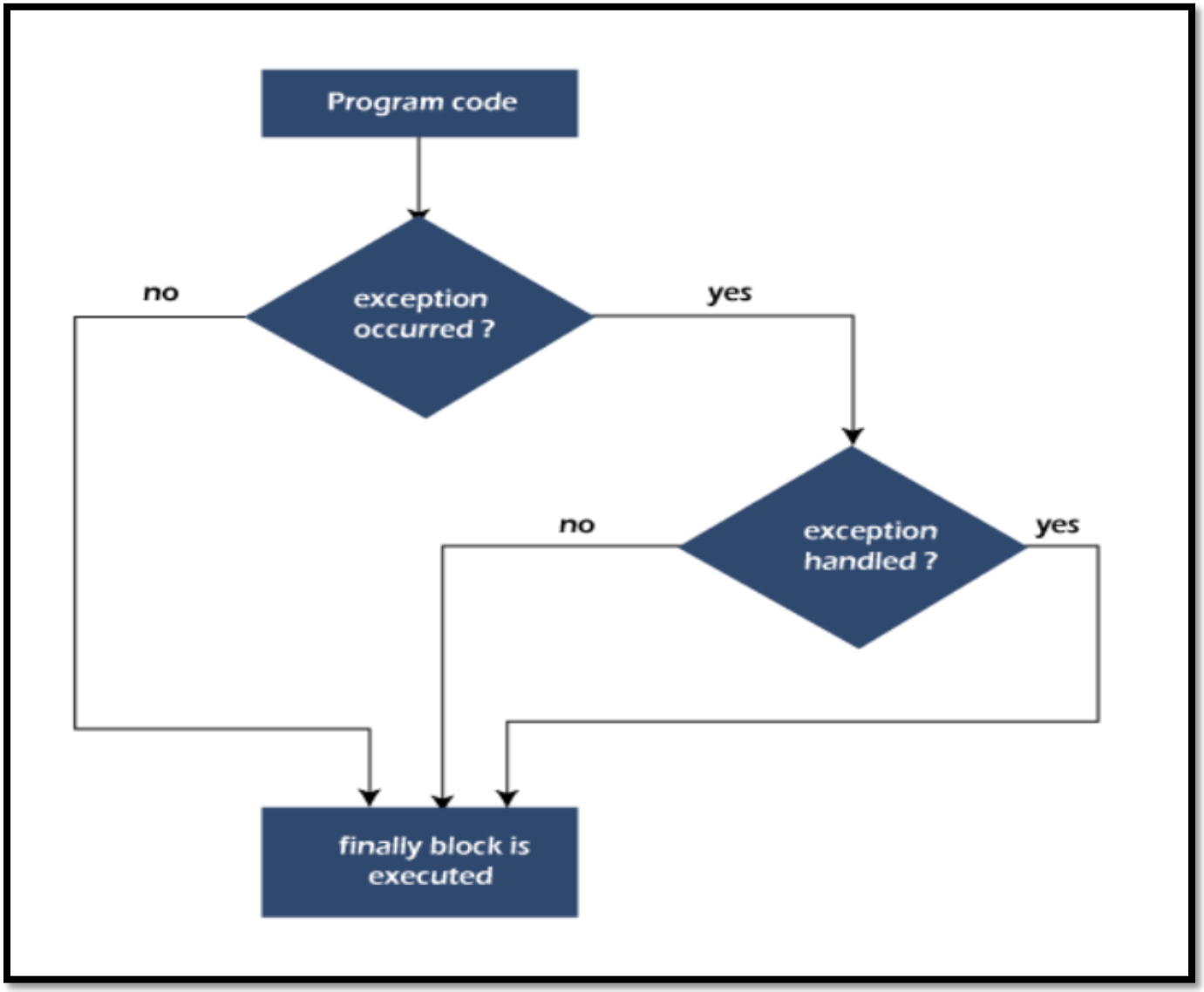
throw v/s throws

throw	throws
Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
throw is used within the method.	throws is used with the method signature.

Finally Statement

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- **finally** block will be executed whether or not an exception is thrown.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- Each try clause requires at least one catch or finally clause.

Finally Statement



Finally Statement

- Finally block in Java can be used to put "cleanup" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.
- There are 3 Different cases when we use Finally keyword, These are when:

- There is no exception occur.
- Exception occur which is handled by the catch block.
- There is an exception occur without handled by the catch block.

Finally Statement

```
try
{
//Protected code
}
catch(ExceptionType1 e1)
{
//Catch block 1
}
catch(ExceptionType2 e2)
{

//Catch block 2
}
finally
{
//The finally block always executes.

}
```

Ex:-When there is no exception occur:

```java
import java.util.*;

public class Main{
    public static void main(String args[]){
        // try block
        try{
            // try block Statements
            int temp = 50/5;
            System.out.println(temp);
            System.out.println("I am in the try Block");
        }
        // catch block
        catch(NullPointerException e){
                        //catch block exceptions
            System.out.println(e);
        }
        // finally block
        finally {
            // finally blick statement
            System.out.println("I am in finally block");
        }
    }
}
```

Output:
10
I am in the try Block
I am in finally block

Ex:-When there is an exception occur which is handled by the catch block :

```java
import java.util.*;

public class Main{
    public static void main(String args[]){
        // try block
        try{
            System.out.println("I am in the try Block");
            int temp = 50/0;
            System.out.println(temp);
        }
        //Catch Block
        catch(ArithmeticException e){
            System.out.println(e);
        }
        //finally block
        finally {
            System.out.println("I am in finally block");
        }
  }
}
```

Output:
I am in the try Block
va.lang.ArithmeticException
/ by zero
I am in finally block

Ex:-When there is an exception occur without handled by the catch block.:

```java
public class Main{
    public static void main(String args[]){
        // try block
        try{
            System.out.println("I am in the try Block");
            int temp = 50/0;
            System.out.println(temp);
        }
        // catch block
        catch(NullPointerException e){
            System.out.println(e);
        }
        // finally block
        finally {
            System.out.println("I am in finally block");
        }
  }
}
```

Output:
I am in the try Block
I am in finally block
Exception in thread "main" java.lang.ArithmeticException / by zero at Prep.main(Prep.java:9)

Example

```
// Demonstrate finally.
class FinallyDemo {
   // Through an exception out of the method.
   static void procA() {
      try {
         System.out.println("inside procA");
         throw new RuntimeException("demo");
      } finally {
         System.out.println("procA's finally");
      }
   }

   // Return from within a try block.
   static void procB() {
      try {
         System.out.println("inside procB");
         return;
      } finally {
         System.out.println("procB's finally");
      }
   }

   // Execute a try block normally.
   static void procC() {
      try {
         System.out.println("inside procC");
      } finally {
         System.out.println("procC's finally");
      }
   }

   public static void main(String args[]) {
      try {
         procA();
      } catch (Exception e) {
         System.out.println("Exception caught");
      }
      procB();
      procC();
   }
}
```

Output

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Finally Statement

- Note:
- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clause for every try/catch.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

User Defined Exception

- In java, we can create our own exception that is known as custom exception or user-defined exception.
- We can have our own exception and message.

Key points to keep in mind:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

User Defined Exception

- It can be useful when we want to create an exception that is specific to our program.
- These can be done by simply extending Java Exception class.
- You can define a constructor for your exception subclass and you can override the tostring() function to display your customized message.

User Defined Exception

- Built-in exception classes handle some generic errors.
- For application-specific errors define your own exception classes. How?
- Define a subclass of Exception:

```
class MyException extends Exception {
        …
}
```

- MyException need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

User Defined Exception

```
class demoUserException extends Exception
{
        private int ex;
        demoUserException(int a)
        {
                ex=a;
        }
        public String toString()
        {
        return "MyException[" + ex +"] is less than zero";
        }
}
class demoException
{
        static void sum(int a,int b) throws demoUserException
        {
        if(a<0)
        {
        throw new demoUserException (a);
        }
        else
        {
        System.out.println(a+b);
        }
}
```

User Defined Exception

```
public static void main(String[] args)
{
        try
        {
        sum(-10, 10);
        }
        catch(demoUserException e)
        {
        System.out.println(e);
        }
}
```

Output:
MyException[-10] is less than zero

User Defined Exception

- For Example:

```
class MyException extends Exception{
   private int a;
   MyException(int i) {
      a = i;
   }
   public String toString (){
      return “MyException[“ + a+”]”;
   }
}
```

Continuation of the Example

```
class test{
   static void compute (int a) throws Myexception{
       if(a>10) throw new MyException(a);
       System.out.println("Normal Exit");
   }

public static void main(String args[]){
   try{
       compute(1);
       compute(20);
       } catch(MyException e){
          System.out.println("Caught " +e);
        }
} //end of class
```

Example-2

```
class InvalidRadiusException extends Exception {
       private double r;
      public InvalidRadiusException(double radius){
             r = radius;
        }
        public void printError(){
           System.out.println("Radius["+ r+"] is not valid");
        }
}
```

Continuation of Example-2

```
class Circle {
 double x, y, r;
 public Circle(double centreX,double centreY,double radius )
throws InvalidRadiusException {
        if (r <= 0 ) {
                throw new InvalidRadiusException(radius);
        }
        else {
                x = centreX ; y = centreY;  r = radius;
        }
    }
}
```

Continuation of Example-2

```
class CircleTest {
  public static void main(String[] args){
      try{
          Circle c1 = new Circle(10, 10, -1);
          System.out.println("Circle created");
        }
        catch(InvalidRadiusException e)
        {
          e.printError();
        }
      }
}
```

Java Built-In Exceptions

- The default java.lang package provides several exception classes, all sub-classing the RuntimeException class.
- Two sets of build-in exception classes:
  - unchecked exceptions – the compiler does not check if a method handles or throws there exceptions
  - checked exceptions – must be included in the method's throws clause if the method generates but does not handle them

Unchecked Built-In Exceptions

- Methods that generate but do not handle those exceptions need not declare them in the throws clause:
  1) ArithmeticException
  2) ArrayIndexOutOfBoundsException
  3) ArrayStoreException
  4) ClassCastException
  5) IllegalStateException
  6) IllegalMonitorStateException
  7) IllegalArgumentException

8. StringIndexOutOfBounds
9. UnsupportedOperationException
10. SecurityException
11. NumberFormatException
12. NullPointerException
13. NegativeArraySizeException
14. IndexOutOfBoundsException
15. IllegalThreadStateException

Checked Built-In Exceptions

- Methods that generate but do not handle those exceptions must declare them in the throws clause:
  1. NoSuchMethodException
  2. NoSuchFieldException
  3. InterruptedException
  4. InstantiationException
  5. IllegalAccessException
  6. CloneNotSupportedException
  7. ClassNotFoundException

List of Java Exception (Built-In Exception)

- Java defines several built-in exception classes inside the standard package **java.lang.**
- **Checked Exception:**

Exception	Description
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program.
IllegalAccessException	This Exception occurs when you create an object of an abstract class and interface.
NotSuchMethodException	This Exception occurs when the method you call does not exist in class.
NoSuchFieldException	A requested field does not exist.

List of Java Exception (Built-In Exception)

- **Unchecked Exception:**

Exception	Description
ArithmeticException	This Exception occurs, when you divide a number by zero causes an Arithmetic Exception.
ArrayIndexOutOfBounds Exception	This Exception occurs, when you assign an array which is not compatible with the data type of that array.
NumberFormatException	This Exception occurs, when you try to convert a string variable in an incorrect format to integer (numeric format) that is notcompatible with each other.
ClassCastException	Invalid cast.
NullPointerException	Invalid use of a null reference.

Java Optional Class

- Java introduced a new class Optional in jdk8.
- It is a public final class and used to deal with NullPointerException in Java application.
- You must import java.util package to use this class.
- It provides methods which are used to check the presence of value for particular variable.
- Optional is a container object used to contain not-null objects.
- Optional object is used to represent null with absent value.
- Declaration for java.util.Optional<T> class –

public final class Optional<T> extends Object

Java Optional Class

```
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        Optional<String> checkNull = Optional.ofNullable(str[5]);
        if(checkNull.isPresent()){  // check for value is present or not
            String lowercaseString = str[5].toLowerCase();
            System.out.print(lowercaseString);
        }else
            System.out.println("string value is not present");
    }
}
```

string value is not present

Java Optional Class

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      checkNull.ifPresent(System.out::println);   // printing value by using method reference
      System.out.println(checkNull.get());    // printing value by using get method
      System.out.println(str[5].toLowerCase());
   }
}
```

```
JAVA OPTIONAL CLASS EXAMPLE
JAVA OPTIONAL CLASS EXAMPLE
java optional class example
```