CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

Lecture 6 – Exceptions handling

Exception

- In Java, runtime errors are thrown as exceptions.
- An exception is an object that represents an error or a condition that prevents execution from proceeding normally.
- If the exception is not handled, the program will terminate abnormally.
- How can you handle the exception so the program can continue to run or else terminate gracefully?

Exception-Handling Overview

- Show runtime error Quotient
- Fix it using an if statement QuotientWithIf
- With a method QuotientWithIf

Exception-Handling Overview
Show runtime error

```
Enter two integers: 1 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:9)
```

```java
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      // Prompt the user to enter two integers
      System.out.print("Enter two integers: ");
      int number1 = input.nextInt();
      int number2 = input.nextInt();
      System.out.println(number1 + " / " + number2 + " is " +(number1 / number2));
   }
}
```

Exception-Handling Overview
Fix it using an if statement

```
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    if (number2 != 0)
      System.out.println(number1 + " / " + number2 + " is " +
          (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

```
Enter two integers: 1 0
Divisor cannot be zero
```

Exception-Handling Overview With a method

```
public class Main {
   public static int quotient(int number1, int number2) {
      if (number2 == 0) {
         System.out.println("Divisor cannot be zero");
         System.exit(1);
      }
      return number1 / number2;
   }
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("Enter two integers: ");
      int number1 = input.nextInt();
      int number2 = input.nextInt();
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "+ result);
}}
```

```
Enter two integers: 1 0
Divisor cannot be zero
```

Exception-Handling Overview With a method

- The method quotient (previous slide) returns the quotient of two integers. If number2 is 0, it cannot return a value, so the program is terminated in line 7.
- This is clearly a problem. You should not let the method terminate the program—the caller should decide whether to terminate the program.
- How can a method notify its caller an exception has occurred? Java enables a method to throw an exception that can be caught and handled by the caller . as shown in next slide.

using exception handling

```
import java.util.Scanner;
public class Main {
   public static int quotient(int number1, int number2) {
      if (number2 == 0)
         throw new ArithmeticException("Divisor cannot be zero");

      return number1 / number2;
   }
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("Enter two integers: ");
      int number1 = input.nextInt();
      int number2 = input.nextInt();
      try {
         int result = quotient(number1, number2);
         System.out.println(number1 + " / " + number2 + " is " + result);
      }
      catch (ArithmeticException ex) {
         System.out.println("Exception: an integer " +"cannot be divided by zero
");
      }
      System.out.println("Execution continues ...");
   }
}
```

```
Enter two integers: 1 0
Exception: an integer cannot be divided by zero
Execution continues ...
```

using exception handling

- The value thrown, in this case new ArithmeticException("Divisor cannot be zero"), is called an exception.
- The execution of a throw statement is called throwing an exception.
- The exception is an object created from an exception class. In this case, the exception class is java.lang.ArithmeticException.
- The constructor ArithmeticException(str) is invoked to construct an exception object, where str is a message that describes the exception.

using exception handling

- When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to “throw an exception” is to pass the exception from one place to another.
- The statement for invoking the method is contained in a try block. The try block contains the code that is executed in normal scinario.
- The exception is caught by the catch block.
- The code in the catch block is executed to handle the exception. Afterward, the statement after the catch block is executed.

using exception handling

- The throw statement is similar to a method call, but instead of calling a method, it calls a catch block.
- In this sense, a catch block is like a method definition with a parameter that matches the type of the value being thrown.
- Unlike a method, however, after the catch block is executed, the program control does not return to the throw statement; instead, it executes the next statement after the catch block.

using exception handling

- The identifier ex in the catch–block header catch (ArithmeticException ex)
- acts very much like a parameter in a method. Thus, this parameter is referred to as a catch block parameter.
- The type (e.g., ArithmeticException) preceding ex specifies what kind of exception the catch block can catch.
- Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.

using exception handling

- In summary, a template for a try-throw-catch block may look as follows:

```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

using exception handling

- An exception may be thrown directly by using a throw statement in a try block, or by invoking a method that may throw an exception.
- The main method invokes quotient (line 20). If the quotient method executes normally, it returns a value to the caller.
- If the quotient method encounters an exception, it throws the exception back to its caller.
- The caller’s catch block handles the exception.

Exception Advantages

- Now you see the *advantages* of using exception handling.
- It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.
- Often the called method does not know what to do in case of error.

Exception V.S If-else

- You *should* use if / else to handle all cases you expect. You should *not* use try {} catch {} to handle *everything* (in most cases) because a useful Exception could be raised and you can learn about the presence of a bug from it.
- You *should* use try {} catch {} in situations where you suspect something can/will go wrong and you don't want it to bring down the whole system, like network timeout/file system access problems, files doesn't exist, etc.

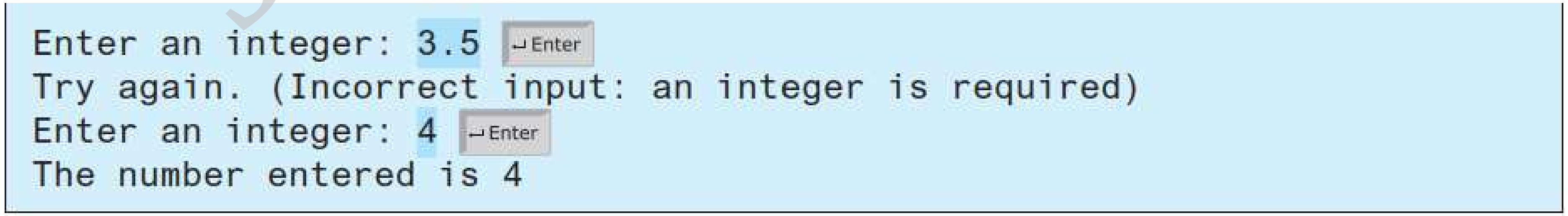
Handling InputMismatchException

InputMismatchExceptionDemo

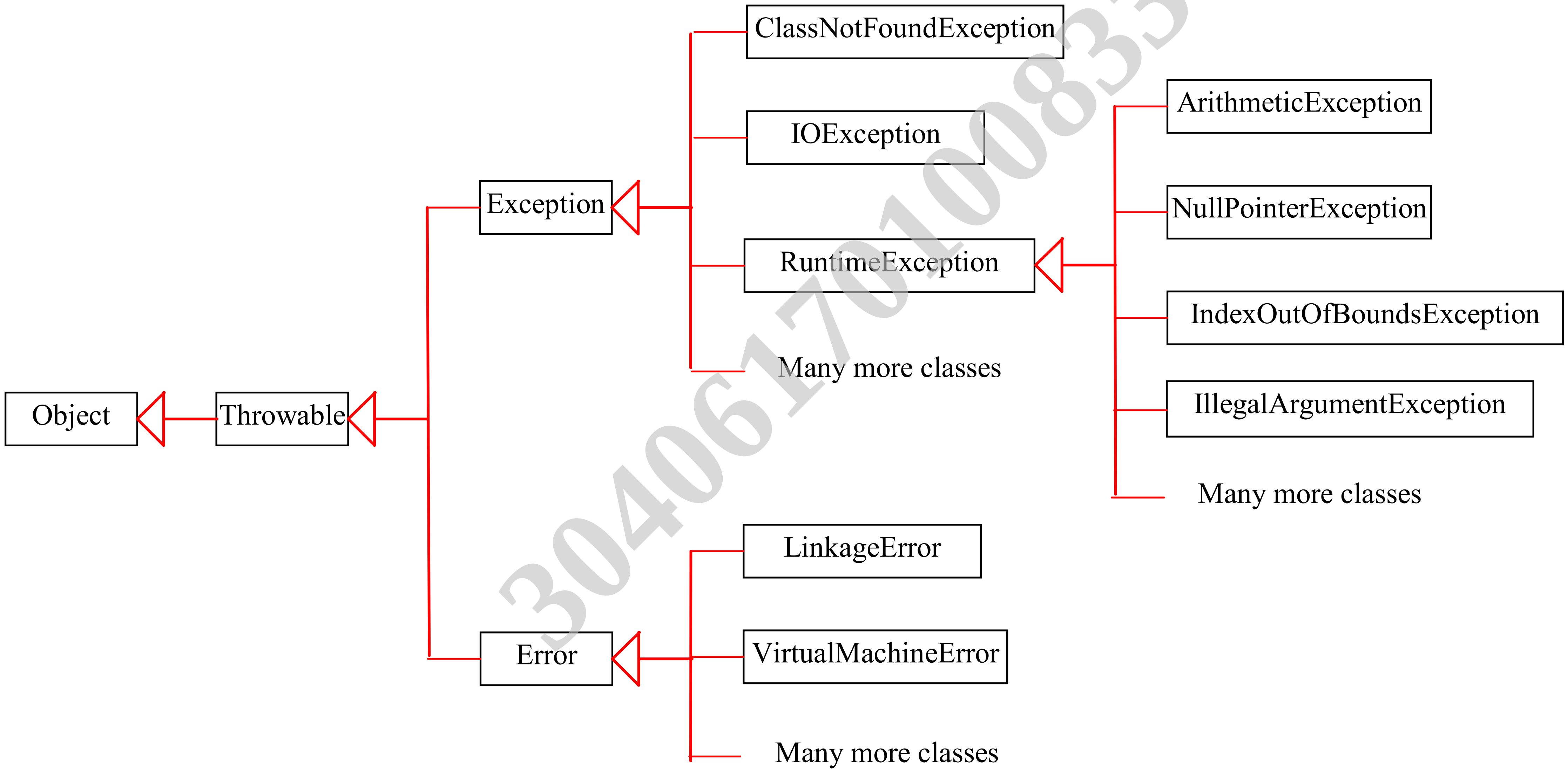
By handling InputMismatchException, your program will continuously read an input until it is correct.

```java
import java.util.*;
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;
    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();
     // Display the result
        System.out.println("The number entered is " + number);
        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```

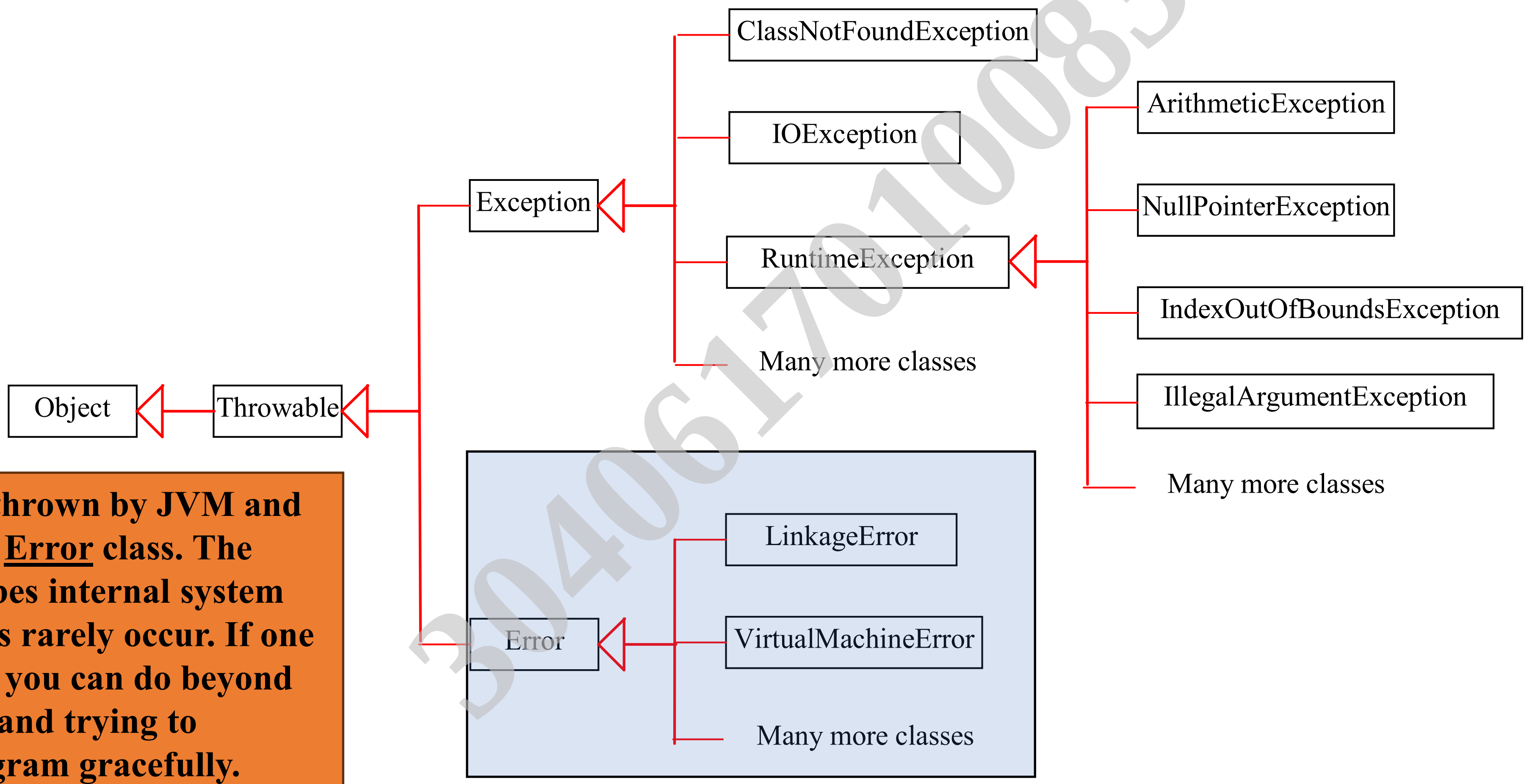
When executing input.nextInt() (line 11), an InputMismatchException occurs if the input entered is not an integer. Suppose 3.5 is entered. An InputMismatchException occurs and the control is transferred to the catch block. The statements in the catch block are now executed. The statement input.nextLine() in line 22 discards the current input line so the user can enter a new line of input. The variable continueInput controls the loop. Its initial value is true (line 6) and it is changed to false (line 17) when a valid input is received. Once a valid input is received, there is no need to continue the input.


```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

Exception Types



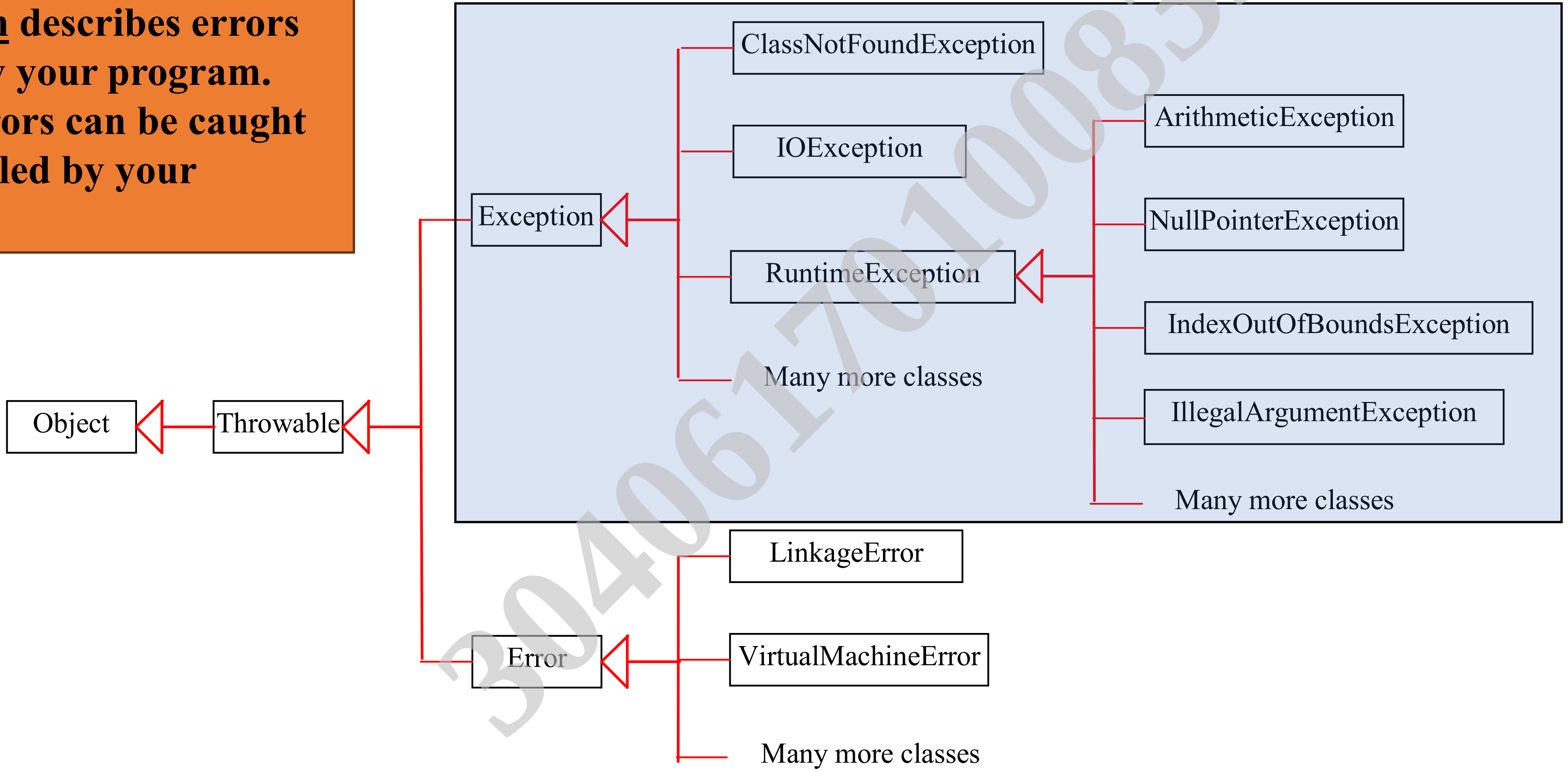
System Errors



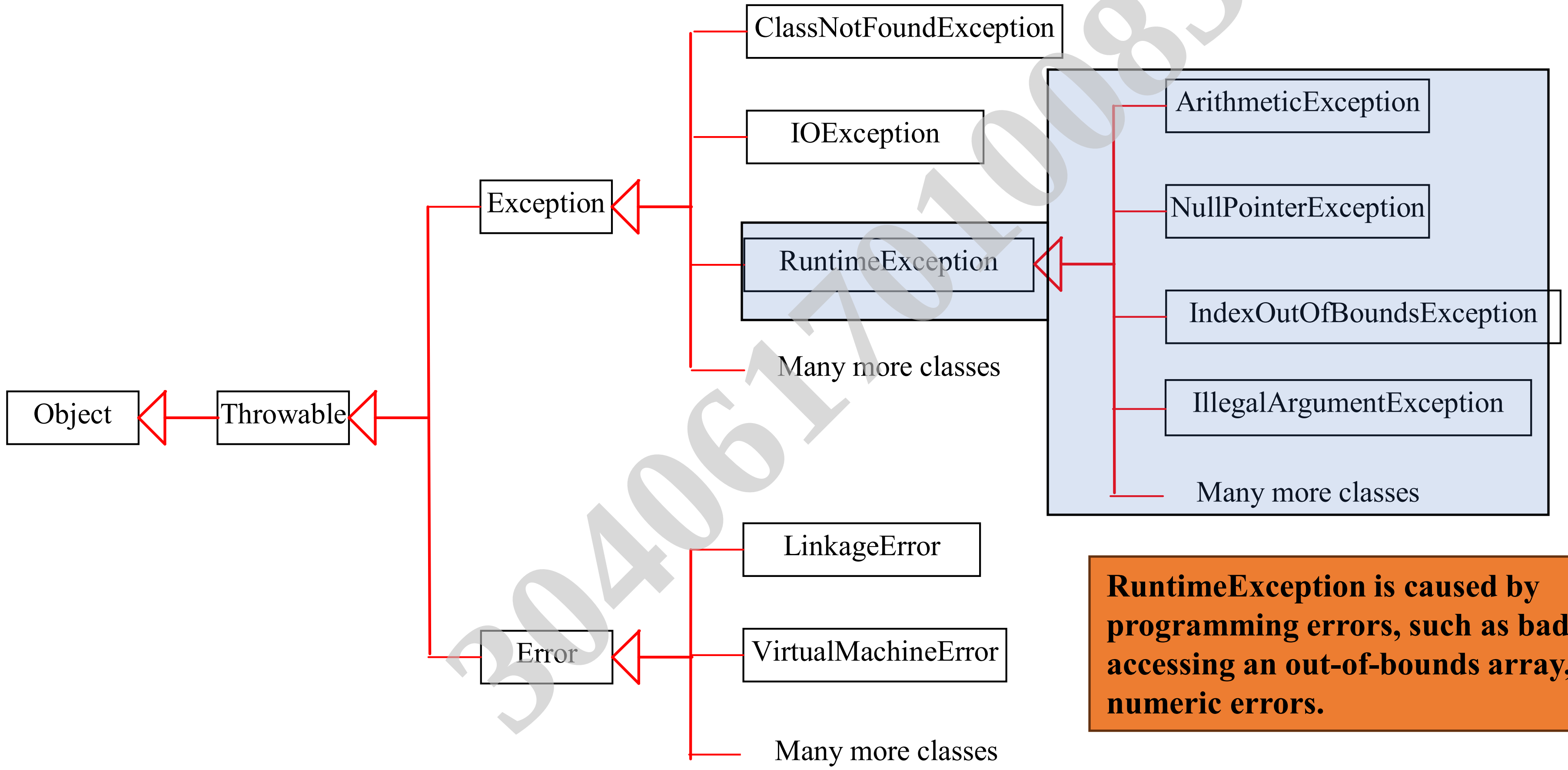
System errors **are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.**

Exceptions

<u>Exception</u> describes errors caused by your program. These errors can be caught and handled by your program.



Runtime Exceptions



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Examples of Subclasses of Exception

- **ClassNotFoundException** Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the java command or if your program were composed of, say, three class files, only two of which could be found.
- **IOException** Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedIOException, EOFException (EOF is short for End of File), and FileNotFoundException.

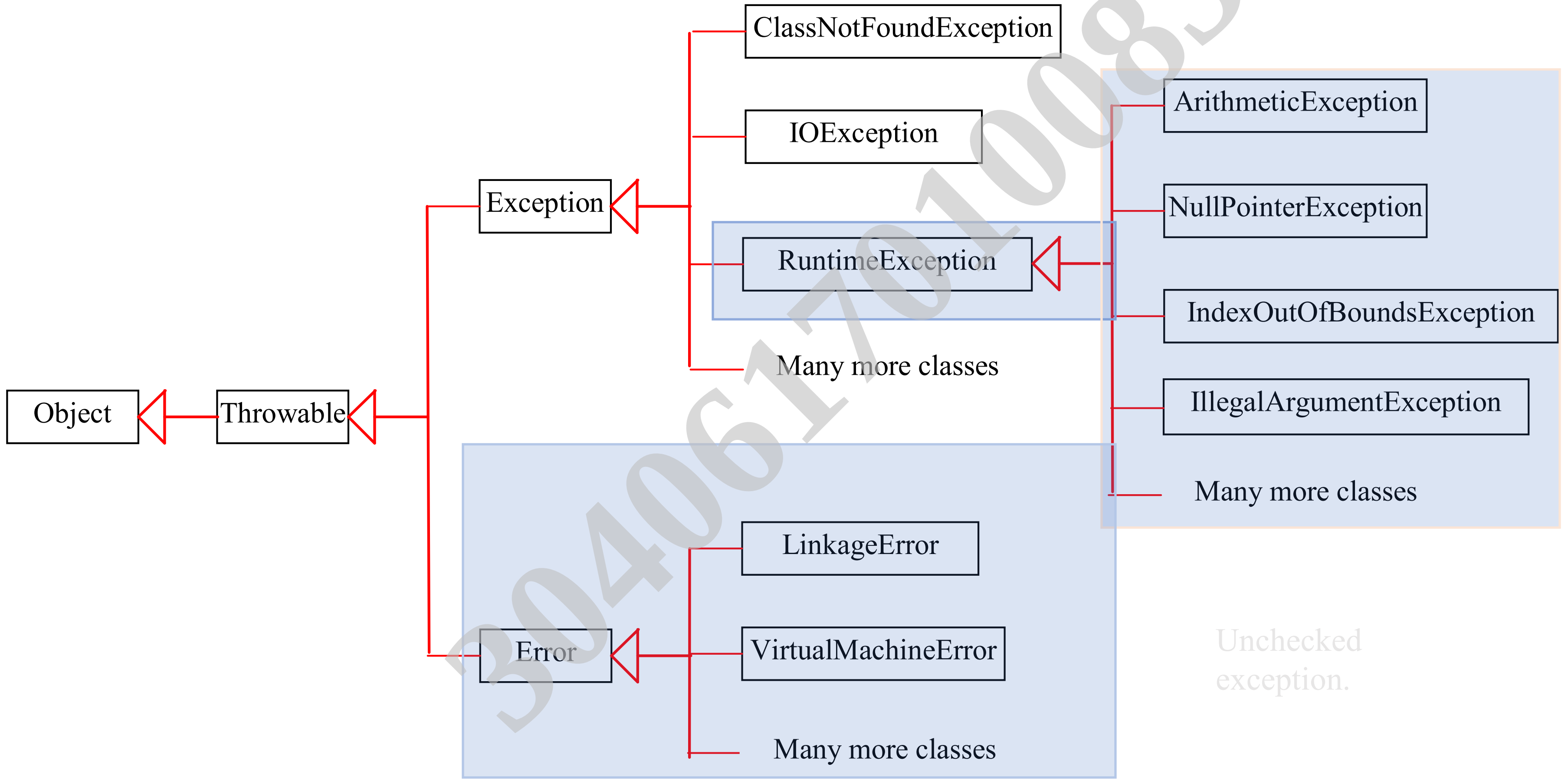
Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

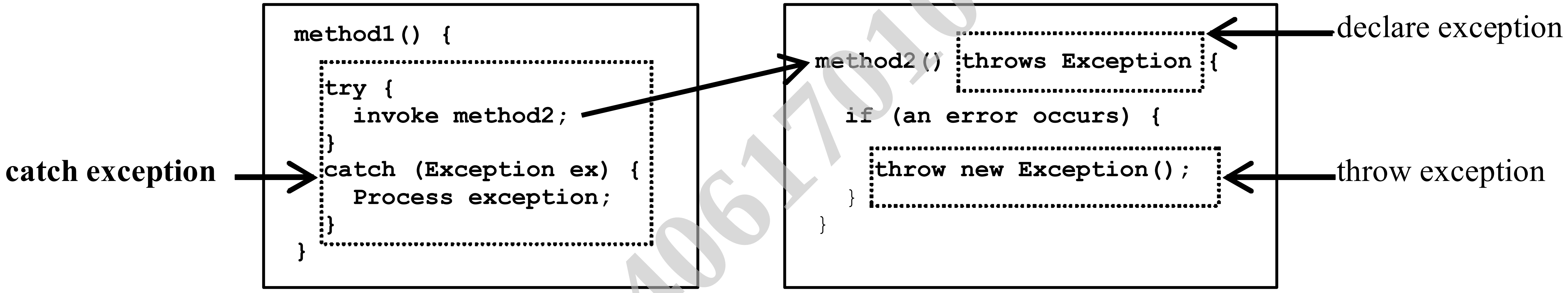
Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
- For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array.
- These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program.
- To avoid overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Unchecked Exceptions



Declaring, Throwing, and Catching Exceptions



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Java's exception-handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception

Declaring Exceptions

- Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.
- Because system errors and runtime errors can happen to any code, Java does not require that you declare Error and RuntimeException (unchecked exceptions) explicitly in the method. However, all other exceptions thrown by the method must be explicitly declared in the method header so the caller of the method is informed of the exception.
- The throws keyword indicates myMethod might throw an IOException.
- **public void myMethod() throws IOException**
- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after throws:**public void myMethod() throws IOException, OtherException,…,Exception**
- **If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.(Liskov Substitution)**

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```

Throwing Exceptions Example

- Here is an example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of IllegalArgumentException and throw it, as follows:

```
IllegalArgumentException ex =
  new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```

Throwing Exceptions Example

```
/** Set a new radius */
 public void setRadius(double newRadius)
      throws IllegalArgumentException {
   if (newRadius >= 0)
      radius =  newRadius;
   else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
 }
```

Note

- IllegalArgumentException is an exception class in the Java API.
- In general, each exception class in the Java API has at least two constructors: a no-arg constructor and a constructor with a String argument that describes the exception.
- This argument is called the exception message, which can be obtained by invoking getMessage() from an exception object.

TIP

- The keyword to declare an exception is throws, and the keyword to throw an exception is throw .

Catching Exceptions

```
try {
   statements;     //   Statements   that   may   throw
  exceptions
}
catch (Exception1 exVar1) {
   handler for exception1;
}
catch (Exception2 exVar2) {
   handler for exception2;
}
...
catch (ExceptionN exVar3) {
   handler for exceptionN;
}
```

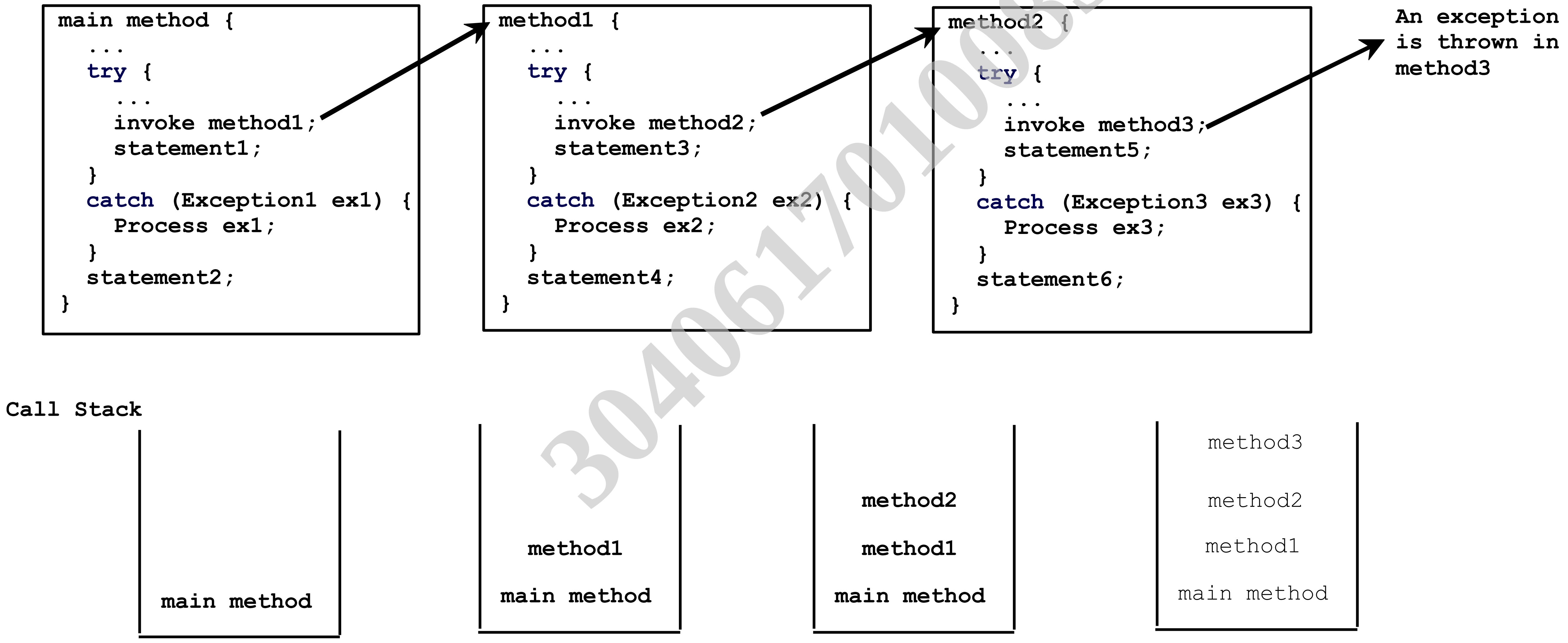
Catching Exceptions

- If one of the statements inside the try block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception.
- The code that handles the exception is called the exception handler; it is found by propagating the exception backward through a chain of method calls, starting from the current method.
- Each catch block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block.

Catching Exceptions

- If so, the exception object is assigned to the variable declared and the code in the catch block is executed.
- If no handler is found, Java exits this method, passes the exception to the method’s caller, and continues the same process to find a handler.
- If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.
- The process of finding a handler is called catching a exception.

Catching Exceptions



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Catching Exceptions

- Suppose the main method invokes method1, method1 invokes method2, method2 invokes method3, and method3 throws an exception, as shown in Figure 12.3. Consider the following scenario:
- ■ If the exception type is Exception3, it is caught by the catch block for handling exception ex3 in method2. statement5 is skipped and statement6 is executed.
- ■ If the exception type is Exception2, method2 is aborted, the control is returned to method1, and the exception is caught by the catch block for handling exception ex2 in method1. statement3 is skipped and statement4 is executed.

Catching Exceptions

- ■If the exception type is Exception1, method1 is aborted, the control is returned to the main method, and the exception is caught by the catch block for handling exception ex1 in the main method. statement1 is skipped and statement2 is executed.
- ■ If the exception type is not caught in method2, method1, or main, the program
- terminates and statement1 and statement2 are not executed.

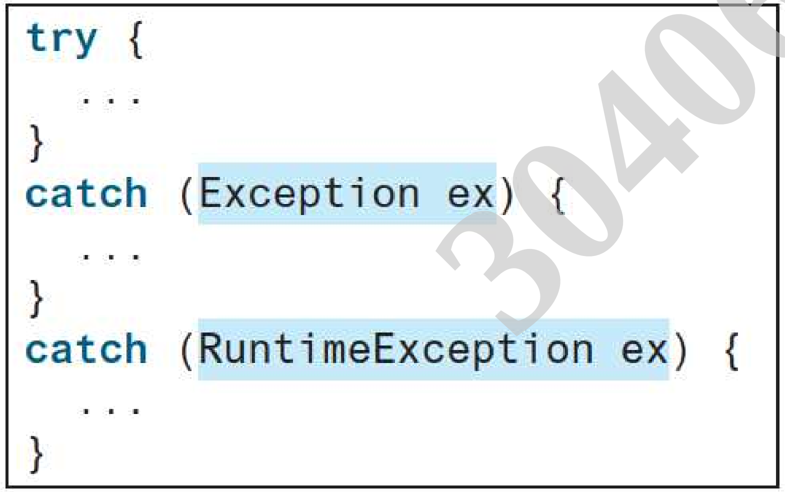
```java
public class Main {
   public static void main(String[] args) {
      try {
         methodA();
      } catch (Exception e) {
         System.out.println("Caught exception in main: " + e.getMessage());
      }
   }
   public static void methodA() throws Exception {
      System.out.println("In methodA");
      methodB();
      System.out.println("This line in methodA will not be executed if exception is thrown.");
   }
   public static void methodB() throws Exception {
      System.out.println("In methodB");
      int res= 5/0;
      System.out.println("This line in methodB will not be executed if exception is thrown.");
 }
}
```

```
In methodA
In methodB
Caught exception in main: / by zero

=== Code Execution Successful ===
```

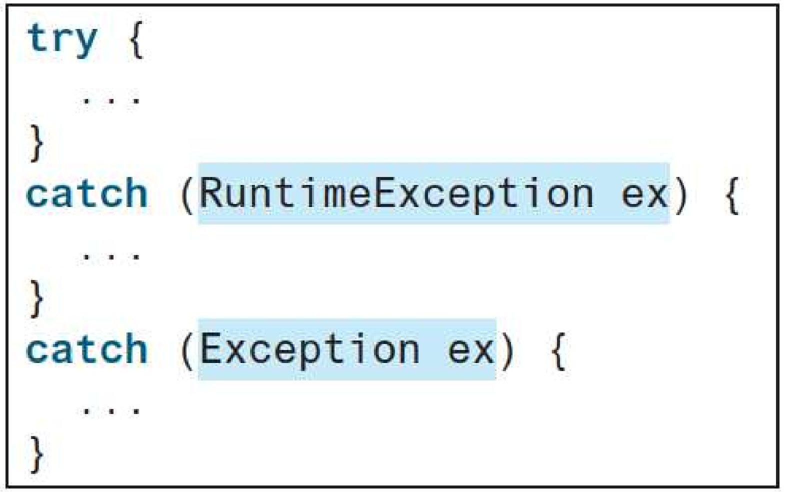
Catching Exceptions

- The order in which exceptions are specified in catch blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because RuntimeException is a subclass of Exception. The correct ordering should be as shown in (b).



```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order



```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

Catching Exceptions

- Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException); you have to write the code as shown in (a) or (b) below.

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a) Catch exception

```
void p1() throws IOException {

  p2();

}
```

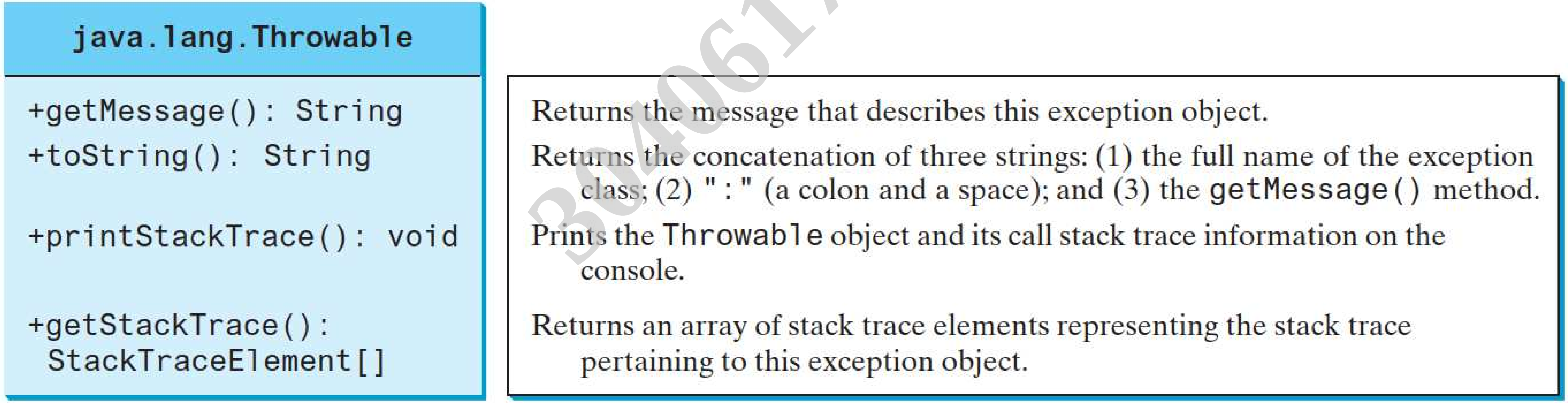
(b) Throw exception

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

Getting Information from Exceptions

- An exception object contains valuable information about the exception. You may use the following instance methods in the java.lang.Throwable class to get information regarding the exception, as shown in Figure 12.4. The printStackTrace() method prints stack trace



java.lang.Throwable	
+getMessage(): String	Returns the message that describes this exception object.
+toString(): String	Returns the concatenation of three strings: (1) the full name of the exception class; (2) ": " (a colon and a space); and (3) the getMessage() method.
+printStackTrace(): void	Prints the Throwable object and its call stack trace information on the console.
+getStackTrace(): StackTraceElement[]	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

```
public class Main {
   public static void main(String[] args) {
      try {
         System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
      }
      catch (Exception ex) {
         ex.printStackTrace();
         System.out.println("\n" + ex.getMessage());
         System.out.println("\n" + ex.toString());
         System.out.println("\nTrace Info Obtained from getStackTrace");
         StackTraceElement[] traceElements = ex.getStackTrace();
         for (int i = 0; i < traceElements.length; i++) {
            System.out.print("method " + traceElements[i].getMethodName());
            System.out.print("(" + traceElements[i].getClassName() + ":");
            System.out.println(traceElements[i].getLineNumber() + ")");
         }
      }
   }
   private static int sum(int[] list) {
      int result = 0;
      for (int i = 0; i <= list.length; i++)
         result += list[i];
      return result;
   }}
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
	at Main.sum(Main.java:23)
	at Main.main(Main.java:4)

Index 5 out of bounds for length 5

java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

Trace Info Obtained from getStackTrace
method sum(Main:23)
method main(Main:4)
```

printStackTrace()printStackTrace()

getMessage()

toString()

Using
getStackTrace()

Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 9. The new setRadius method throws an exception if radius is negative.

CircleWithException

TestCircleWithException

```
public class CircleWithException {
   /** The radius of the circle */
   private double radius;
   /** The number of the objects created */
   private static int numberOfObjects = 0;
   /** Construct a circle with radius 1 */
   public CircleWithException() {
      this(1.0);
   }
   /** Construct a circle with a specified radius */
   public CircleWithException(double newRadius) {
      setRadius(newRadius);
      numberOfObjects++;
   }
   /** Return radius */
   public double getRadius() {
      return radius;
   }
```

```
   public void setRadius(double newRadius)
         throws IllegalArgumentException {
      if (newRadius >= 0)
         radius = newRadius;
      else
         throw new IllegalArgumentException(
            "Radius cannot be negative");
   }
   /** Return numberOfObjects */
   public static int getNumberOfObjects() {
      return numberOfObjects;
   }
   /** Return the area of this circle */
   public double findArea() {
      return radius * radius * 3.14159;
   }
}
```

```
public class Main {

    public static void main(String[] args) {
        try {
            CircleWithException c1 = new CircleWithException(5);
            CircleWithException c2 = new CircleWithException(-5);
            CircleWithException c3 = new CircleWithException(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }

        System.out.println("Number of objects created: " +
                CircleWithException.getNumberOfObjects());
    }
}
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

The `finally` Clause

```
try {
   statements;
}
catch(TheException ex) {
   handling ex;
}
finally {
   finalStatements;
}
```

The `finally` Clause

- The code in the finally block is executed under all situations, regardless of whether an exception occurs in the try block or is caught. Consider three possible cases:
- 1. If no exception arises in the try block, finalStatements is executed and the next statement after the try statement is executed.
- 2. If a statement causes an exception in the try block that is caught in a catch block, the rest of the statements in the try block are skipped, the catch block is executed, and the finally clause is executed. The next statement after the try statement is executed.
- 3. If one of the statements causes an exception that is not caught in any catch block, the other statements in the try block are skipped, the finally clause is executed, and the exception is passed to the caller of this method.

Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception.

```
try {
   statements;
}
catch(TheException ex) {
   perform operations before exits;
   throw ex;
}
```

```java
public class RethrowExample {
    public static void main(String[] args) {
        try {
            process();
        } catch (Exception e) {
            System.out.println("Caught in main: " + e.getMessage());
        }
    }
    public static void process() throws Exception {
        try {
            riskyOperation();
        } catch (Exception e) {
            System.out.println("Caught in process: " + e.getMessage());
            // Rethrow the exception
            throw e;
        }
    }
    public static void riskyOperation() throws Exception {
        throw new Exception("Error occurred in riskyOperation");
    }
}
```

```
//output
Caught in process: Error occurred in riskyOperation
Caught in main: Error occurred in riskyOperation
```

Explanation:

1.riskyOperation throws an exception.

2.The process method catches it, logs the error, and rethrows the same exception.

3.The main method catches the rethrown exception and handles it.

animation

Trace a Program Execution

Suppose no exceptions in the statements

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

animation

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

When to Throw Exceptions

- An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

- When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
   System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
   System.out.println("refVar is null");
}
```

When to Use Exceptions

- is better to be replaced by

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
```

Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

Custom Exception Class Example

- In Listing 13.8, the setRadius method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler, you have to create a custom exception class.

InvalidRadiusException

CircleWithRadiusException

TestCircleWithRadiusException

```java
public class InvalidRadiusException extends Exception {
    private double radius;
    public InvalidRadiusException(double radius) {
        super("Invalid radius " + radius);
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
}
```

```
public class CircleWithRadiusException {
    private double radius;
    private static int numberOfObjects = 0;
  public CircleWithRadiusException() {
    this(1.0);
  }
  public CircleWithRadiusException(double newRadius) {
    try {
      setRadius(newRadius);
      numberOfObjects++;
    }
    catch (InvalidRadiusException ex) {
      ex.printStackTrace();
    }
  }
public double getRadius() {
    return radius;
  }
```

```
public void setRadius(double newRadius)
      throws InvalidRadiusException {
    if (newRadius >= 0)
      radius = newRadius;
    else
      throw new InvalidRadiusException(newRadius);
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

```java
public class Main {
   public static void main(String[] args) {
      try {
         CircleWithRadiusException c1 = new CircleWithRadiusException(5);
         c1.setRadius(-5);
         CircleWithRadiusException c3 = new CircleWithRadiusException(0);
      }
      catch (InvalidRadiusException ex) {
         System.out.println(ex);
      }

      System.out.println("Number of objects created: " +
            CircleWithRadiusException.getNumberOfObjects());
   }
}
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```

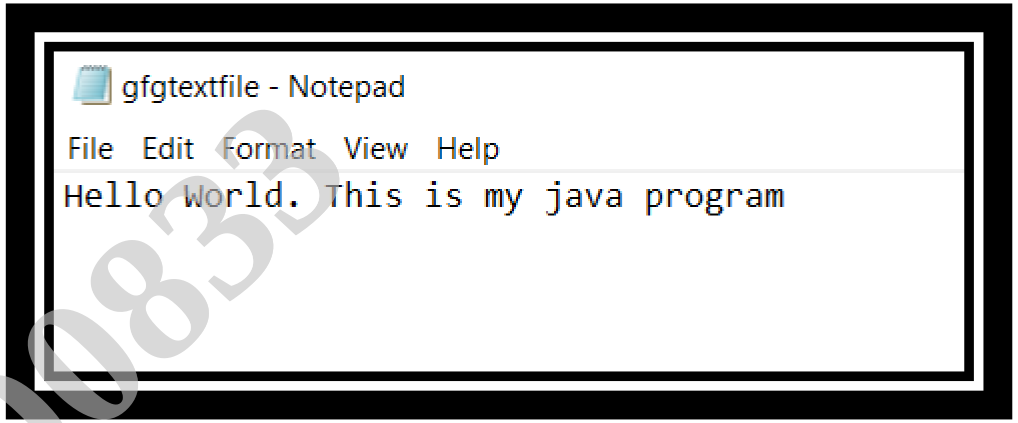
Try-with-resources Feature in Java

- In Java, the Try-with-resources statement is a try statement that declares one or more resources in it. A resource is an object that must be closed once your program is done using it. For example, a File resource or a Socket connection resource. The try-with-resources statement ensures that each resource is closed at the end of the statement execution. If we don't close the resources, it may constitute a resource leak and also the program could exhaust the resources available to it.
- By this, now we don't need to add an extra finally block for just passing the closing statements of the resources. The resources will be closed as soon as the try-catch block is executed.

Syntax: Try-with-resources

```
try(declare resources here) {
    // use resources
}
catch(FileNotFoundException e) {
    // exception handling
}
```

Output:
```
Resource are closed and message has been
written into the gfgtextfile.txt
```


gfgtextfile - Notepad

```
Hello World. This is my java program
```

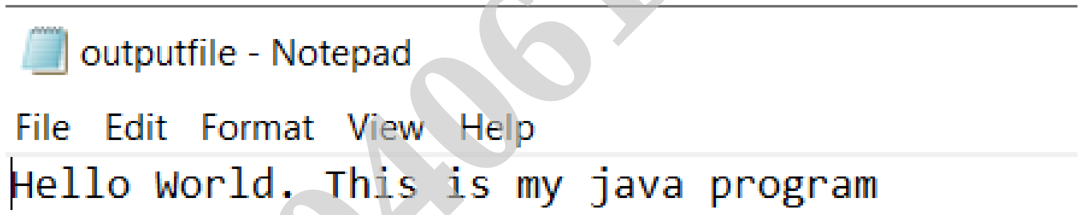
```
import java.io.*;
class GFG {
	public static void main(String[] args)
	{
		// Try block to check for exceptions
		try (FileOutputStream fos = new FileOutputStream("gfgtextfile.txt")) {
			String text = "Hello World. This is my java program";
			byte arr[] = text.getBytes();
			fos.write(arr);
		}
		catch (Exception e) {
			System.out.println(e);
		}
		System.out.println(
		"Resource are closed and message has been written into the
		gfgtextfile.txt");
	}
}
```

```
import java.io.*;
class GFG {
public static void main(String[] args)
      {
            try (FileOutputStream fos = new FileOutputStream("outputfile.txt");
                 BufferedReader br =
                   new BufferedReader( new FileReader("gfgtextfile.txt"))
                   )
                   {
                   while ((text = br.readLine()) != null) {
                          byte arr[] = text.getBytes();
                   }

                   System.out.println( "File content copied to another one.");
            }
            catch (Exception e) {
                   System.out.println(e);
            }
            System.out.println(
                   "Resource are closed and message has been written into the
                   gfgtextfile.txt");
      }
}
```

Output:

```
File content copied to another one. Resource
are closed and message has been written into
the gfgtextfile.txt
```



```
Hello World. This is my java program
```

Thanks

References

- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- This slides based on slides provided by Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- © Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.
- php - What is the advantage of using try {} catch {} versus if {} else {} - Stack Overflow