Lab 11
Exception Handling

Objective

- To get an overview of exceptions and exception handling.
- To distinguish exception types: Error (fatal) vs. Exception (nonfatal) and checked vs. unchecked.
- To declare exceptions in a method header.
- To throw exceptions in a method.
- To write a try-catch block to handle exceptions.
- To use the finally clause in a try-catch block.

Introduction

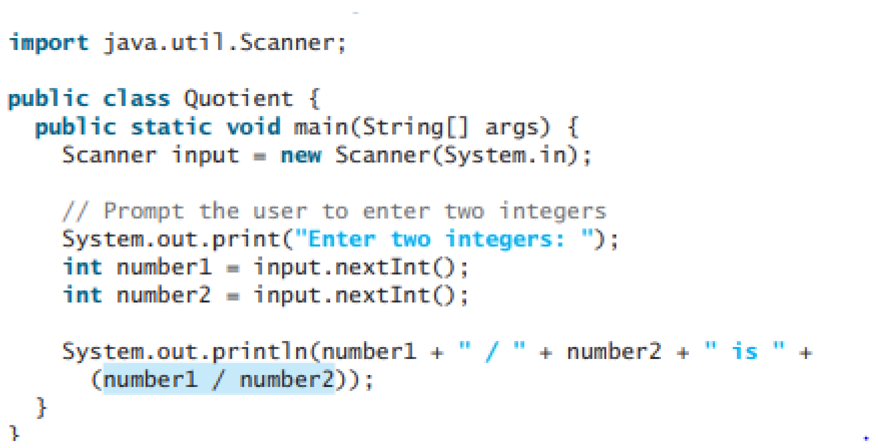
Exception handling enables a program to deal with exceptional situations and continue its normal execution. Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an `ArrayIndexOutOfBoundsException`. If you enter a double value when your program expects an integer, you will get a runtime error with an `InputMismatchException`.
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 that the program can continue to run or else terminate gracefully?

template for a try-throw-catch block may look like this:

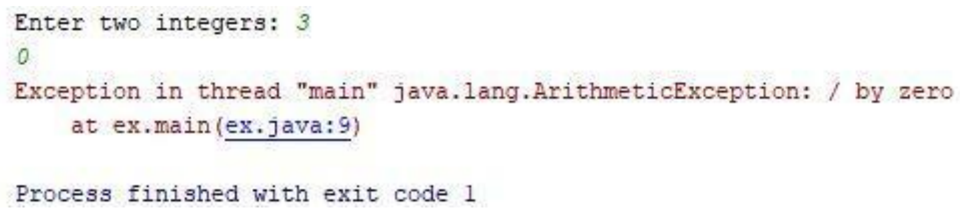
```
try {
Code to run;
A statement or a method that may throw an exception;
More code to run;
 }
catch (type ex) {
Code to process the exception;
}
```

Exception-Handling Overview

Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

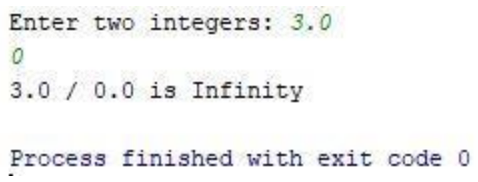


```
import java.util.Scanner;

public class Quotient {
  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));
  }
}
```



```
Enter two integers: 3
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ex.main(ex.java:9)

Process finished with exit code 1
```

Note that a floating-point number divided by 0 does not raise an exception.



```
Enter two integers: 3.0
0
3.0 / 0.0 is Infinity

Process finished with exit code 0
```

With Method:

```
import java.util.Scanner;
public class QuotientWithMethod {
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);
// Prompt the user to enter two integers
```

```
 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 " 23        + result);
 }
}
```

Output:

```
Enter two integers: 5 0
Divisor cannot be zero
```

This is clearly a problem. You should not let the method terminate the program—the caller should decide whether to terminate the program.

With Exception Handling:

```
import java.util.Scanner;
   public class QuotientWithException {
       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);
     // Prompt the user to enter two integers
        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 ...");
```

```
  }
 }
```

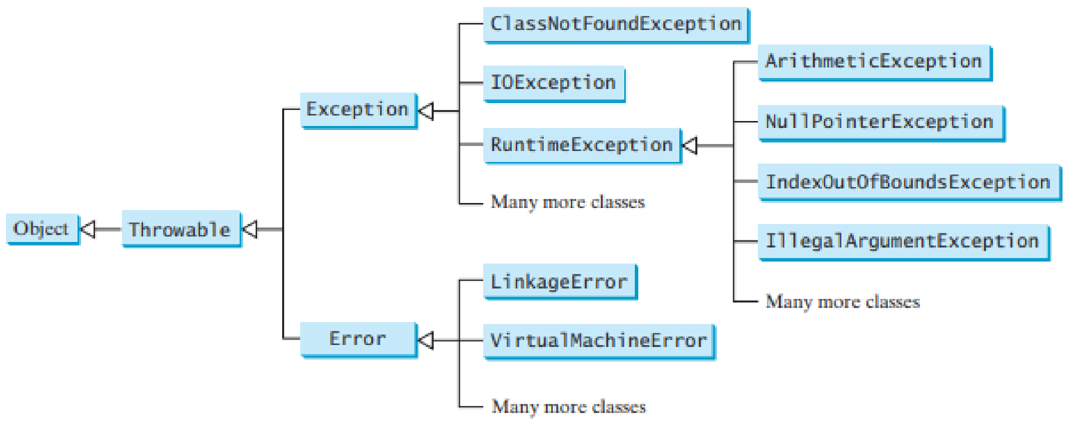
Output:

```
Enter two integers: 5 0
Exception: an integer cannot be divided by zero
Execution continues ...
```

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.

Exception Types

Exceptions are objects, and objects are defined using classes. The root class for exceptions is java.lang.Throwable. The preceding section used the classes ArithmeticException and InputMismatchException



Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.

Declaring Exceptions:

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the main method to start executing a program. 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 that the caller of the method is informed of the exception. To declare an exception in a method, use the throws keyword in the method header, as in this example:

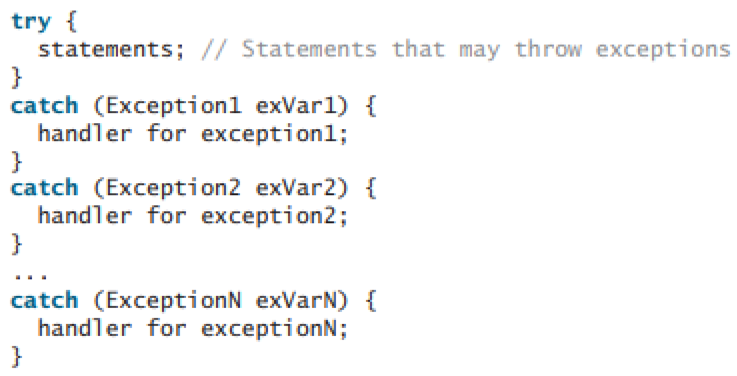
public void myMethod() throws IOException

The throws keyword indicates that myMethod might throw an IOException. If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after throws:

public void myMethod() throws Exception1, Exception2, ..., ExceptionN

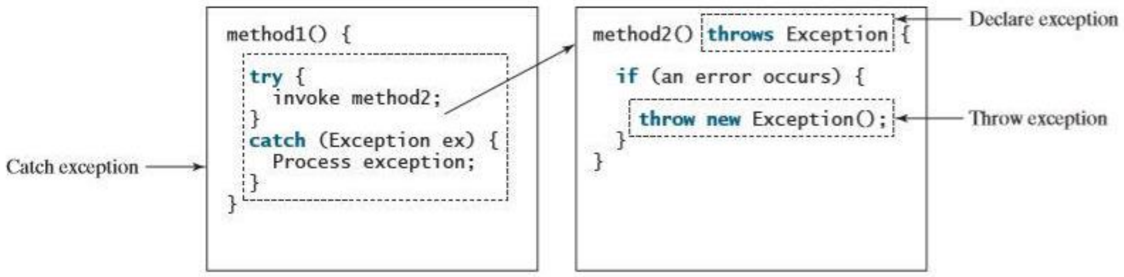
Catching Exceptions

When an exception is thrown, it can be caught and handled in a try-catch block, as follows:



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

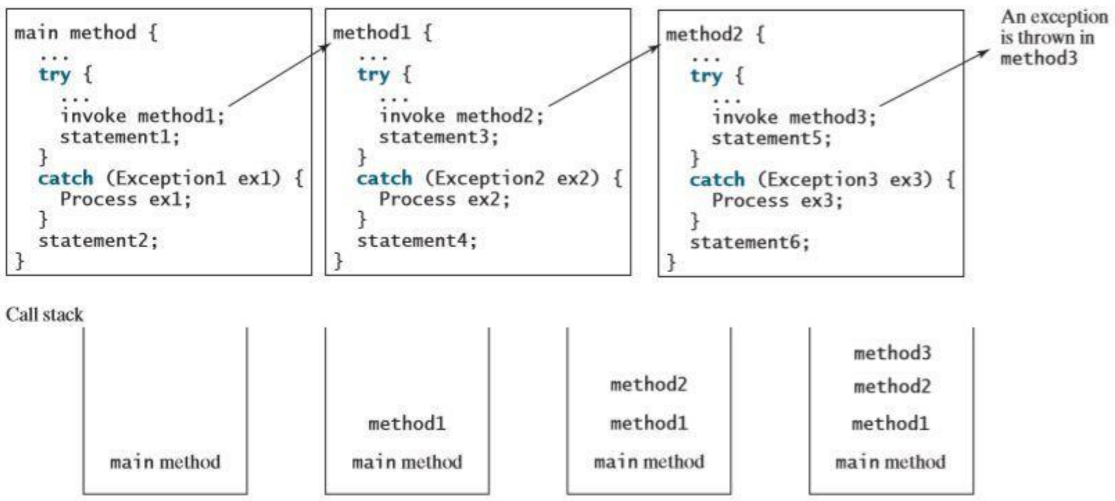
If no exceptions arise during the execution of the try block, the catch blocks are skipped. If one of the statements inside the try block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception. The code that handles 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. 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 that invoked the method, 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 an exception



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

FIGURE 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing 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;
}
```

FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

The finally Clause

The finally clause is always executed regardless whether an exception occurred or not. Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a `finally` clause that can be used to accomplish this objective. The syntax for the `finally` clause might look like this:

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


finally {
  finalStatements;
}
```

The code in the finally block is executed under all circumstances, regardless of whether an exception occurs in the try block or is caught.
Consider three possible cases:

- If no exception arises in the try block, final Statements is executed, and the next statement after the try statement is executed.
- 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.
- 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. The finally block executes even if there is a return statement prior to reaching the finally block.

To Get Exception Information:

```
System.out.println(ex.toString());
System.out.println(ex.getMessage());
```

Practice Task

Task 1:

What RuntimeException will the following programs throw, if any?

```
1. public class Test {
        public static void main(String[] args) {
        System.out.println(1 / 0);
       }
       }
```

```
2. public class Test {
                        public static void main(String[] args) {
                        int[] list = new  int[5];
                       System.out.println(list[5]);
                       } }

3. public class Test {
                       public static void main(String[] args) {
                       String s = "abc";
                       System.out.println(s.charAt(3));
                       } }

4. public class Test { public static void main(String[] args) {
                     Object o = new Object();
                     String d = (String)o;
                      } }

5. public class Test {
                     public static void main(String[] args) {
                     Object o = null;
                     System.out.println(o.toString());
                     } }

6. public class Test {
                    public static void main(String[] args) {
                    System.out.println(1.0 / 0);  } }
```