SWEG3101

Object Oriented Programming

Software Engineering Department

AASTU

Chapter 6

Exception Handling

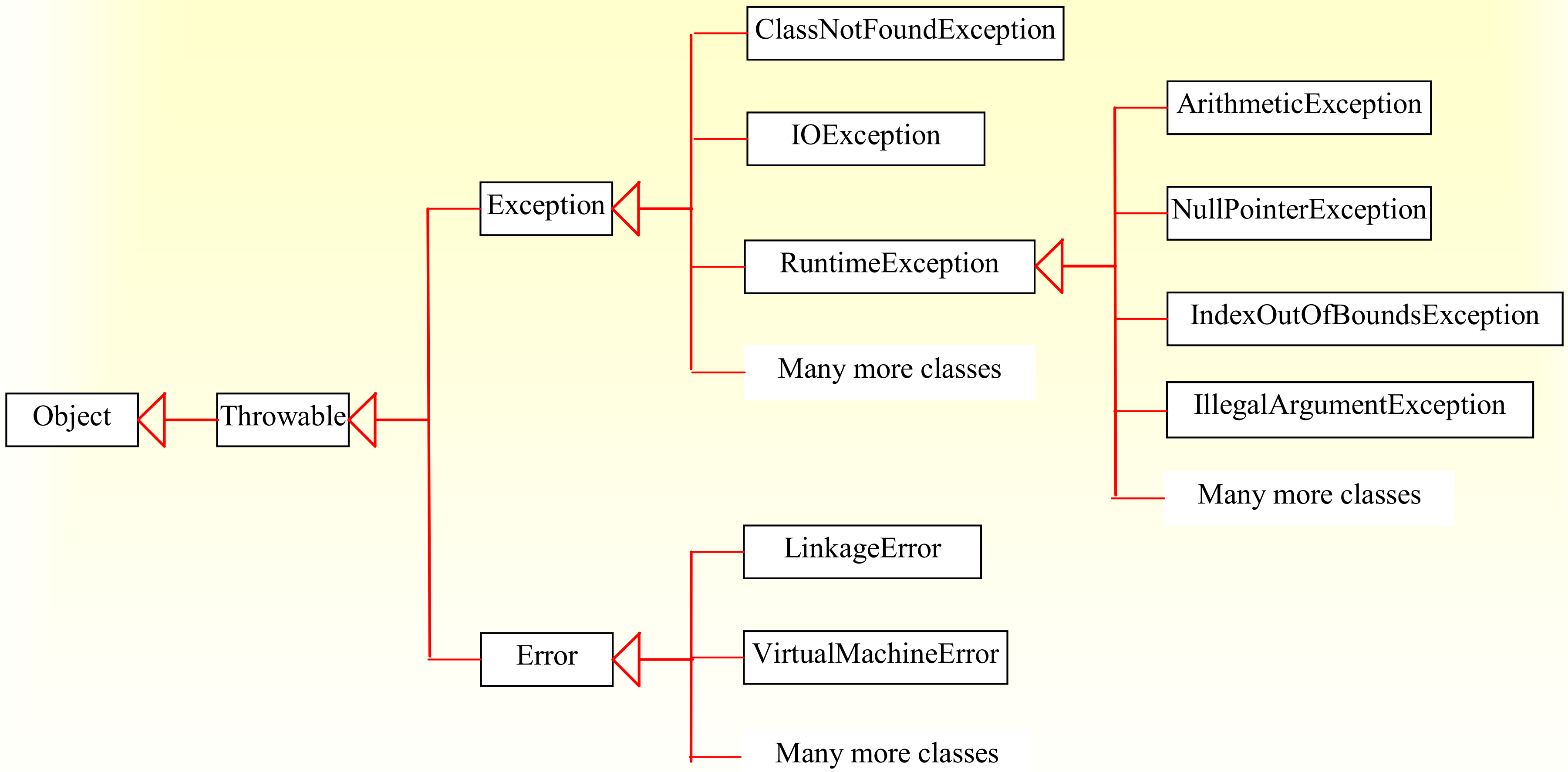
Objectives

- Describe the notion of exception handling
- To throw and catch exceptions
- Use Java’s exception-handling facilities effectively in your own classes and programs

Introduction

- An exception is an object that represents an error or a condition that prevents execution from proceeding normally.
- An exception can occur for many different reasons:
  - 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
  - Physical limitations. Disks can fill up; you can run out of available memory
  - Device errors. Remote server unavailable
- Handling an exception allows a program to continue executing **as if no problem had been encountered**.
  - return to safe state, save work, exit gracefully

Exception Types



Exception Types(cont'd)

- The exception classes can be classified into two major types: system error and exception.
- **System errors** are thrown by the JVM and represented in the *Error* class.
  - rarely occur.

Class	Possible Reason for Exception
LinkageError	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
VirtualMachineError	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exception Types(cont'd)

- **Exception** are represented in the Exception class, which describes errors caused by your program and by external circumstances.

Class	Possible Reason for Exception
ClassNotFoundException	Attempt to use a class that does not exist.
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file.

Exception Types(cont'd)

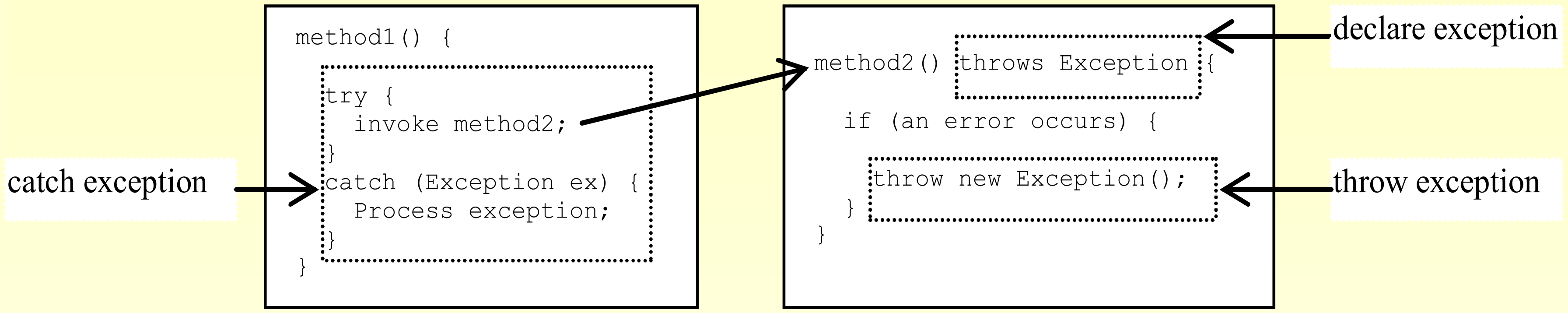
Runtime exceptions are represented in the RuntimeException class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.
ClassNotFoundException	Class not found.
InstantiationException	Attempt to create an object of an abstract class or interface.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

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.
- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array.

Declaring, Throwing, and Catching Exceptions



```
method1() {

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

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

try

Contain program statements that you want to monitor for exceptions

catch

The code in the **catch** block is executed to *handle the exception*

throw

Used to throw exception

throws

used in method heading to declare an exception

finally

Holds code that absolutely must be executed before a method returns

A finally block of code always executes, whether or not an exception has occurred.

Cont'd

- Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions.

```
public Type Method_Name(Parameter_List) throws List_Of_Exceptions
Body_Of_Method
```

```
public void myMethod() throws IOException
```

- 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.

```
throw new TheException();

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

Cont’d

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");
  }
```

Cont'd

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;
}
```

Exception handling overview

- This is the general form of an exception-handling block:

```
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
  }
```

Example: Divide by Zero without Exception Handling

```
import java.util.Scanner;
public class DivideByZeroNoExceptionHandling {
    public static int quotient(int numerator,int denominator ){
        return numerator / denominator;
    }// end method quotient
    public static void main( String args[] ){
        Scanner scanner =new Scanner( System.in ); // scanner for input
        System.out.print("Please enter an integer numerator: ");
        int numerator = scanner.nextInt();
        System.out.print("Please enter an integer denominator: ");
        int denominator = scanner.nextInt();
        int result = quotient( numerator, denominator );
        System.out.print( numerator + " / " + denominator + " = " + result );
    }
}
```

Output

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Example 1

A method catches an exception using a combination of the **try** and **catch** keywords

```
import java.util.Scanner;
public class QuotientWithException {
  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 {
      if (number2 == 0)
       throw new ArithmeticException("Divisor cannot be zero");
       System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
   }
   catch (ArithmeticException ex) {
     System.out.println("Exception: an integer cannot be divided by zero ");
   }
 }
 }
```

Example 2

```
public class CircleWithException {
    private double radius;
    // Construct a circle with a specified radius
    public CircleWithException(double newRadius) {
          setRadius(newRadius);
     }
     /** Return radius */
     public double getRadius() {
         return radius;
     }
```

Example 2(cont’d)

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

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

Example 2(cont'd)

```
public class TestCircleWithException {
     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);
           }
     }
}
```

Output:

```
java.lang. IllegalArgumentException : Radius cannot be negative
```

Example 3

```
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
```

The finally Keyword

- The finally keyword is used to create a block of code that follows a try block.
- A finally block of code always executes, whether or not an exception has occurred.

Example

```
public class ExcepTest{
  public static void main(String args[]){
    int a[]=new int[2];
    try{
      System.out.println("Access element three :"+ a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown :"+ e);
    }
    finally{
      a[0]=6;
      System.out.println("First element value: "+a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This would produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
First element value:6
The finally statement is executed
```

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");
}
```

is better to be replaced by

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