Chapter 10

Exception Handling

Declaration

- These slides are made for UIT, BU students only. I am not holding any copy write of it as I had collected these study materials from different books and websites etc. I have not mentioned those to avoid complexity.

Topics

- Types of errors
  - Compile time error
  - Run time error
- Working of exception handing mechanism
- General form of exception handling
- Place of dealing with errors
- Exception handling
- Other Error Handling Techniques
- The Basics of Java Exception Handling
- Exception Types

Types of errors

- Errors may be broadly classified into two categories
  - Compile time error
  - Run time error

Compile time error

- All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as Compile time errors.
- Whenever compiler displays an error, it will not create the .class file.

Run time error

- Sometimes, a program may compile successfully creating a .class file but may not run properly.
- Such programs may produce wrong results due to wrong logic or may terminate due to errors.
- An exception is an abnormal condition that arises in a code sequence at run time or an exception is a run time error.

Run time error

- Common run time errors are
  - Division by zero
  - Out of bounds array subscript
  - Trying to store a value into an array of an incompatible class or type.
  - Trying to cast an instance of a class to one of its subclass
  - Invalid method parameters
  - Trying to illegally change the state of a thread.
  - Attempting to use a negative size for an array.
  - Using a null object reference as a legitimate object reference to access a method or a variable.
  - Converting invalid string to a number.
  - Accessing a character that is out of bound of a string

  and many more…..

Working of exception handing mechanism

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

- Java exception handling is managed via five keywords:
  1. try
  2. catch
  3. throw
  4. throws
  5. finally

General form of 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
  }
```

Place of dealing with errors

- Errors can be dealt with at place error occurs
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- Exception handling
  - Makes clear, robust, fault-tolerant programs
  - Java removes error handling code from "main line" of program

Exception handling

- Exception handling
  - Catch errors before they occur
  - Deals with synchronous errors (i.e., divide by zero)
  - Does not deal with asynchronous errors
    - Disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
    - Error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly

Exception handling

- Exception handling
  - Should not be used for program control
    - Not optimized, can harm program performance
  - Improves fault-tolerance
    - Easier to write error-processing code
    - Specify what type of exceptions are to be caught
  - Another way to return control from a function or block of code
- Most programs support only single threads

Other Error Handling Techniques

- Other techniques
  - Ignore exceptions
    - Personal programs usually ignore errors
    - Not for commercial software
  - Abort
    - Fine for most programs
    - Inappropriate for mission critical programs

The Basics of Java Exception Handling

- Exception handling
  - Method detects error it cannot deal with
    - *Throws* an exception
  - Exception handler
    - Code to *catch* exception and handle it
  - Exception only caught if handler exists
    - If exception not caught, block terminates

The Basics of Java Exception Handling

- Format
  - Enclose code that may have an error in **`try`** block
  - Follow with one or more **`catch`** blocks
    - Each **`catch`** block has an exception handler
  - If exception occurs and matches parameter in **`catch`** block
    - Code in catch block executed
  - If no exception thrown
    - Exception handling code skipped
    - Control resumes after **`catch`** blocks

```
try{
  code that may throw exceptions

}

catch (ExceptionType ref) {
   exception handling code
}
```

Exception Types

- All exception types are subclasses of the build-in class Throwable.
- Under Throwable there are two subclasses.
- One branch is headed by exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own exception types.
- There is an important subclass of exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Exception Types

- The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type Error are used by the java run time system to indicate errors having to do with the run time environment, itself.
- Stack overflow is an example of this type of error.

Uncaught exceptions

- You should first see when you don't handle them.

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

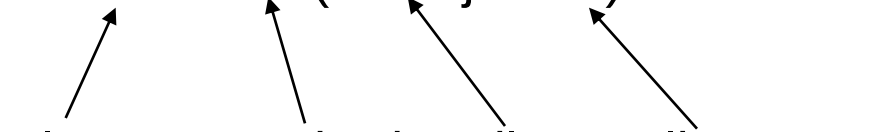
- When a java run time detects the attempt to divide by zero, it construct a new exception object and then throws this exception.
- This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Uncaught exceptions

- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the java run time system.
- JDK run time interpreter will give following output

Java.lang.ArithmeticException: / by zero

at Exc0.main(Exc0.java:4)



Class name — Method name — File name — line number

- All these are included into simple stack trace.

Using try and catch

- Although the default exception handler provided by the java run-time system is useful for debugging, you will usually want to handle an exception yourself
- Benefits are
  - It allows you to fix the error.
  - It prevents the program from automatically terminating.

Using try and catch

```
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.");
  }
}
```

Using try and catch

- Notice that, catch is not “called”, so exception never “returns” to the try block from a catch.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/ catch mechanism.
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if error had never happened.

Using try and catch

```
import java.util.Random;
class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    } } }
```

Displaying a description of an Exception

- Throwable (superclass of all classes handles exceptions) overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

- Output

```
Exception:java.lang.ArithmaticException:/ by zero
```

Multiple catch clauses

```
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.");
  }
}
```

Multiple catch clauses

- Input

C:\> java MultiCatch

- Output

a=0

Divide by 0:java.lang.ArithmaticException:/ by zero

After try/catch blocks.

- Input

C:\> java MultiCatch TestArg

a=1

Array index oob: java.lang.ArrayIndexOutOfBoundException

After try/catch blocks.

Multiple catch clauses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses: Thus a subclass would never be reached if it come after its superclass.

/* 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.

***/**

Multiple catch clauses

```
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.
- If an inner try statement does not have a catch handler for a particular exception, this stack is unwound and the nest try statement's catch handlers are inspected for a catch.

```
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);
```

Nested try Statements

```
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
```

Nested try Statements

```
/* 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);
    }
  }
}
```

Nested try Statements

- Input

C:\> java NestTry

- Output

Divide by 0:java.lang.ArithmaticException:/ by zero

- Input

C:\> java NestTry One

- Output

a=1

Divide by 0:java.lang.ArithmaticException:/ by zero

- Input

C:\> java NestTry One Two

- Output

a=2

Array index out-of-bounds;

oob: java.lang.ArrayIndexOutOfBoundException.

throw

- So far, you have only been catching exceptions that are thrown by the java run time system.
- However, it is possible for your program to throw an exception explicitly using throw statement.
- Here ThrowableInstance must be an object of type Throwable or a subclass of Throwable
- Simple type such as int and char, as well as non - Throwable classes such as String and object, cannot be used as exceptions.
- Throwable object can be obtain by two ways:
  - Using a parameter into catch clause.
  - Creating one with the new operator.

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);
    }
  }
}
```

throw

- Output

Caught inside demoproc.
Recaught: java.lang.NullPointerException:demo.

- Here, new is used.
- All of java's build in run time exceptions have two constructors: one with no parameter and one that takes a string parameter.

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.
- You can do this by including throws clause in the method's declaration.
- If they are not, a compile time error will result.

throws

```
// 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();
  }
}
```

throws

```
// 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);
    }
  }
}
```

finally

- 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 . The finally block will execute whether or not an exception is thrown.

```
// 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");
    }
  }
```

finally

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

finally

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

finally

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

finally

- Output

inside procA

procA's finally

Exception caught

inside procB

procB's finally

inside procC

procC's finally

Creating Your Own Exception Subclasses

- You can do this by creating a subclass of Exception.
- The Exception class does not define any methods of its own.
- It inherit methods of Throwable.

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return "MyException[" + detail + "]";
  }
}
```

Creating Your Own Exception Subclasses

```
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    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);
    }
  }
}
```

Creating Your Own Exception Subclasses

- Output

Called compute (1)

Normal exit

Called compute (20)

Caught MyException[20];

End of Chapter 10

Questions?