Course Name:

Advanced Java

Lecture 5
Topics to be covered

- Exception Handling

Exception Handling-Introduction

- An *exception* is an abnormal condition that arises in a code sequence at run time
- 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
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Exception Handling-Fundamentals

- 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
- System generated exceptions are automatically thrown by the Java run-time system
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause

Exception-Handling Fundamentals

- Any code that absolutely must be executed before a method returns is put in a **finally** block
- 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
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - Error (to handle exceptional conditions that are not expected to be caught under normal circumstances). i.e. stack overflow

Uncaught Exceptions

- If an exception is not caught by user program, then execution of the program stops and it is caught by the default handler provided by the Java run-time system
- Default handler prints a stack trace from the point at which the exception occurred, and terminates the program

Ex:

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

Output:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
Exception in thread "main"
```

Using try and catch

- Handling an exception has two benefits,
  - It allows you to fix the error
  - It prevents the program from automatically terminating
- The **catch** clause should follow immediately the **try** block
- Once an exception is thrown, program control transfer out of the **try** block into the catch block
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism

Example

```java
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.
```

Using try and catch

- A **try** and **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement

```
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<10; 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);
    }
  }
}
```

Multiple catch Clauses

- If more than one 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

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.");
  }
}
```

Example (Cont.)

- If no command line argument is provided, then you will see the following output:

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks
```

- If any command line argument is provided, then you will see the following output:

```
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```