9-Exception Handling

Overview

This chapter examines Java's exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) 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. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. 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. Any code that absolutely must be executed before a method returns is put in a **finally** block.

Syntax of try-catch-finally

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

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

This program generates the following output:

```
Division by zero.
After catch statement.
```

Example

```
// Handle an exception and move on.
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);
    }
  }
}
```

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. 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, and execution continues after the **try**/**catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
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.");
  }
}
```

This program will cause a division-by-zero exception if it is started with no command-line parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

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

C:\\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.
```

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 came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* 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.
*/
```

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

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

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. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// 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");
    }
  }

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

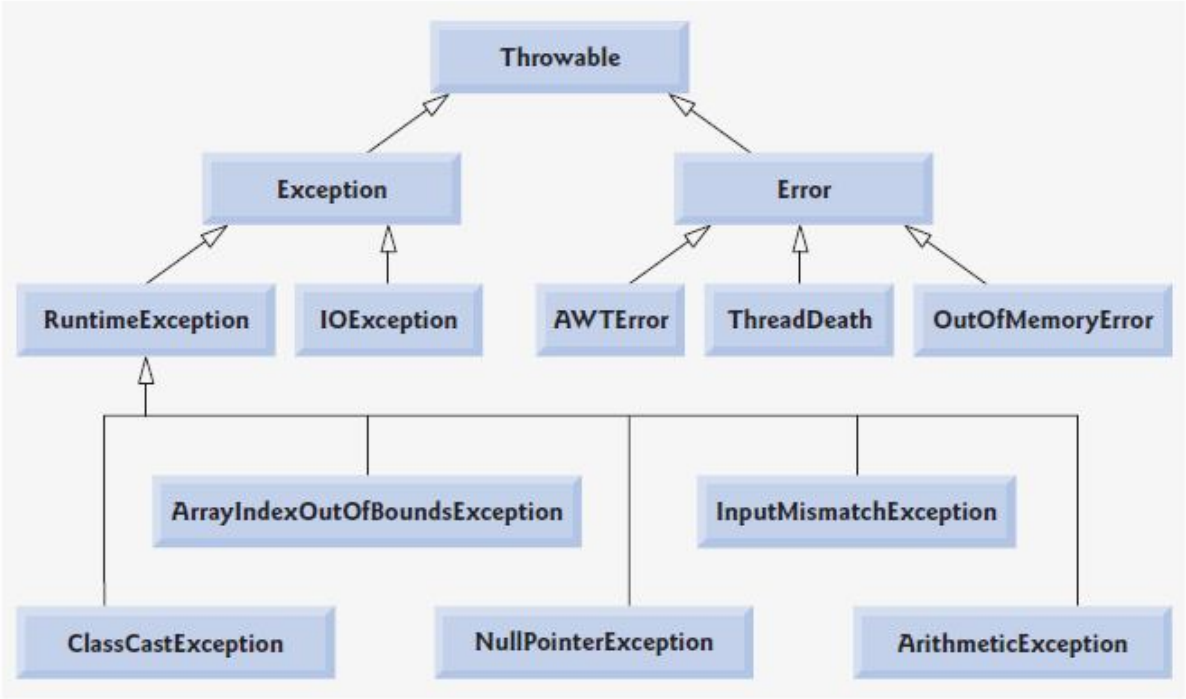
```
      System.out.println("procB's finally");
    }
  }

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

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

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Note If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```



Portion of class `Throwable`'s inheritance hierarchy.