

# Java

## Chapter 10: Exception Handling

Slides material compiled from  
Java - The Complete Reference 9<sup>th</sup> Edition

By  
Herbert Schildt

# Exception-Handling Fundamentals

- **Exception:** an abnormal condition that arises in a code sequence at run time/ run time error
  - Java exception is an object that describes an exceptional (that is, error) condition
  - When an exceptional condition arises, an object representing that exception is created and **thrown** in the method that caused the error
- Exception can be
  - generated by the Java run-time system (relate to fundamental errors that violate the rules of the Java language)
  - Manually generated (typically used to report some error condition to the caller of a method)

# Keywords for exception handling

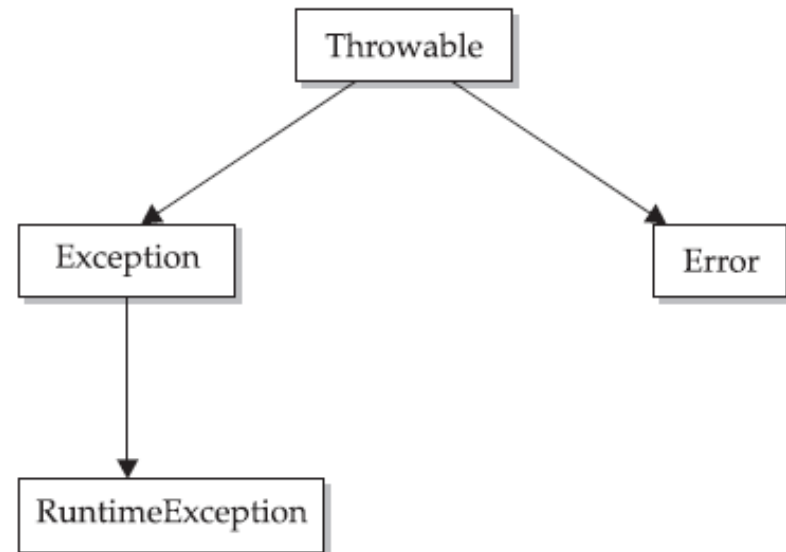
- **try** block contains program statements that are to be monitored for exceptions
  - If an exception occurs within the **try** block, it is thrown
- **catch** block contains statements that catch this exception and handle it in some rational manner
  - System-generated exceptions are automatically thrown by the Java runtime system.
- **throw** is used to manually throw an exception
- **throws** clause is used to specify any exception that is thrown out of a method
- **finally** block contains code that absolutely must be executed after a **try** block completes

# 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 after try block ends  
}
```

# Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Below **Throwable** are two subclasses that partition exceptions into two branches
  - **Exception** class: used for exceptional conditions that user programs should catch
    - RuntimeException etc are subclasses of Exception
  - **Error** class: used by the Java runtime system to indicate errors having to do with the run-time environment (eg Stack overflow)



# Uncaught Exceptions

- Uncaught exception :caught by the **default handler** (provided by the Java run-time system) that
  - displays a string describing the exception
  - prints a stack trace from the point at which the exception occurred
  - terminates the program

**java.lang.ArithmeticException: / by zero  
at Exc0.main(Exc0.java:4)**

# Using try and catch

- Exception handling benefits
  - allows you to fix the error
  - prevents the program from automatically terminating

```
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

```
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.

# Another 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);
        }
    }
}
```

# Displaying a Description of an Exception

- **Throwable** 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.ArithmeticException: / by zero

# Multiple catch Clauses

- Use when more than one exception could be raised by a single piece of code
- 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
  - All others are bypassed
  - execution continues after the **try / catch** block
- NOTE: exception subclasses must come before any of their superclasses because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses

# Multiple catch Clauses (Example)

```
class MultipleCatches {  
    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 (Example)

## OUTPUT

```
C:\>java MultipleCatches
```

```
a = 0
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
```

```
C:\>java MultipleCatches TestArg
```

```
a = 1
```

```
Array index oob:
```

```
java.lang.ArrayIndexOutOfBoundsException:42
```

```
After try/catch blocks.
```

# Nested try Statements

- a **try** statement can be inside the block of another **try**
- Each time a **try** statement is entered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted
- If no **catch** statement matches, then the Java run-time system will handle the exception

# Nested try Statements (Example)

```
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
            try { // nested try block
                if(a==1) a = a/(a-a); // one command-line arg, division by zero
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // two command-line args, 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 (Example)

## OUTPUT

```
C:\>java NestTry
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One
```

```
a = 1
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One Two
```

```
a = 2
```

```
Array index out-of-bounds:
```

```
java.lang.ArrayIndexOutOfBoundsException:42
```

# throw

- To throw an exception explicitly, use the **throw** statement
  - flow of execution stops immediately after the **throw** statement
  - nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception and control is transferred to that statement
  - No match: next enclosing **try** statement is inspected, and so on
  - no matching **catch** found in any block: default exception handler halts the program and prints the stack trace

# throw (example)

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

# throw (example)

```
class ThrowDemo {  
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        } catch(NullPointerException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        } catch(NullPointerException e) {  
            System.out.println("Recaught: " + e);  
        }  
    }  
}
```

## OUTPUT

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

# throws

- If method can cause an exception that it does not handle
  - Then method's declaration must include a **throws** clause (that lists the types of exceptions that a method might throw)
  - Necessary for all exceptions, except **Error** or **RuntimeException**, or any of their subclasses

# throws (example)

```
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);  
        }  
    }  
}
```

# throws (example)

```
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

# 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
  - **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
  - useful for closing file handles and freeing up any other resources
  - **finally** clause is optional

# finally (example)

```
class FinallyDemo {
    static void procA() {
        try {
            System.out.println("inside procA"); throw new RuntimeException("demo");
        } finally { System.out.println("procA's finally");}
    }
    static void procB() {
        try {
            System.out.println("inside procB"); return;
        } finally { System.out.println("procB's finally"); }
    }
    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();
    }
}
```

# finally (example)

```
class FinallyDemo {
    static void procA() {
        try {
            System.out.println("inside procA"); throw new RuntimeException("demo");
        } finally { System.out.println("procA's finally");}
    }
    static void procB() {
        try {
            System.out.println("inside procB"); return;
        } finally { System.out.println("procB's finally"); }
    }
    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();
    }
}
```

## OUTPUT

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

# Java's Built-in Exceptions

Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

| Exception                       | Meaning                                                           |
|---------------------------------|-------------------------------------------------------------------|
| 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.                                                     |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value.         |
| 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.                                      |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of a string.                  |
| TypeNotPresentException         | Type not found.                                                   |
| UnsupportedOperationException   | An unsupported operation was encountered.                         |

# Java's Built-in Exceptions

Java's Checked Exceptions Defined in **java.lang**

| Exception                    | Meaning                                                                            |
|------------------------------|------------------------------------------------------------------------------------|
| ClassNotFoundException       | Class not found.                                                                   |
| CloneNotSupportedException   | Attempt to clone an object that does not implement the <b>Cloneable</b> interface. |
| IllegalAccessException       | Access to a class is denied.                                                       |
| InstantiationException       | Attempt to create an object of an abstract class or interface.                     |
| InterruptedException         | One thread has been interrupted by another thread.                                 |
| NoSuchFieldException         | A requested field does not exist.                                                  |
| NoSuchMethodException        | A requested method does not exist.                                                 |
| ReflectiveOperationException | Superclass of reflection-related exceptions.                                       |