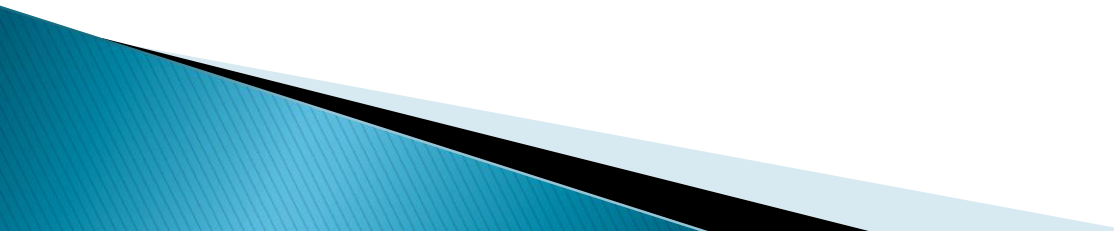


Java Programming

Chapter 07 - Exceptions and Assertions

Objectives

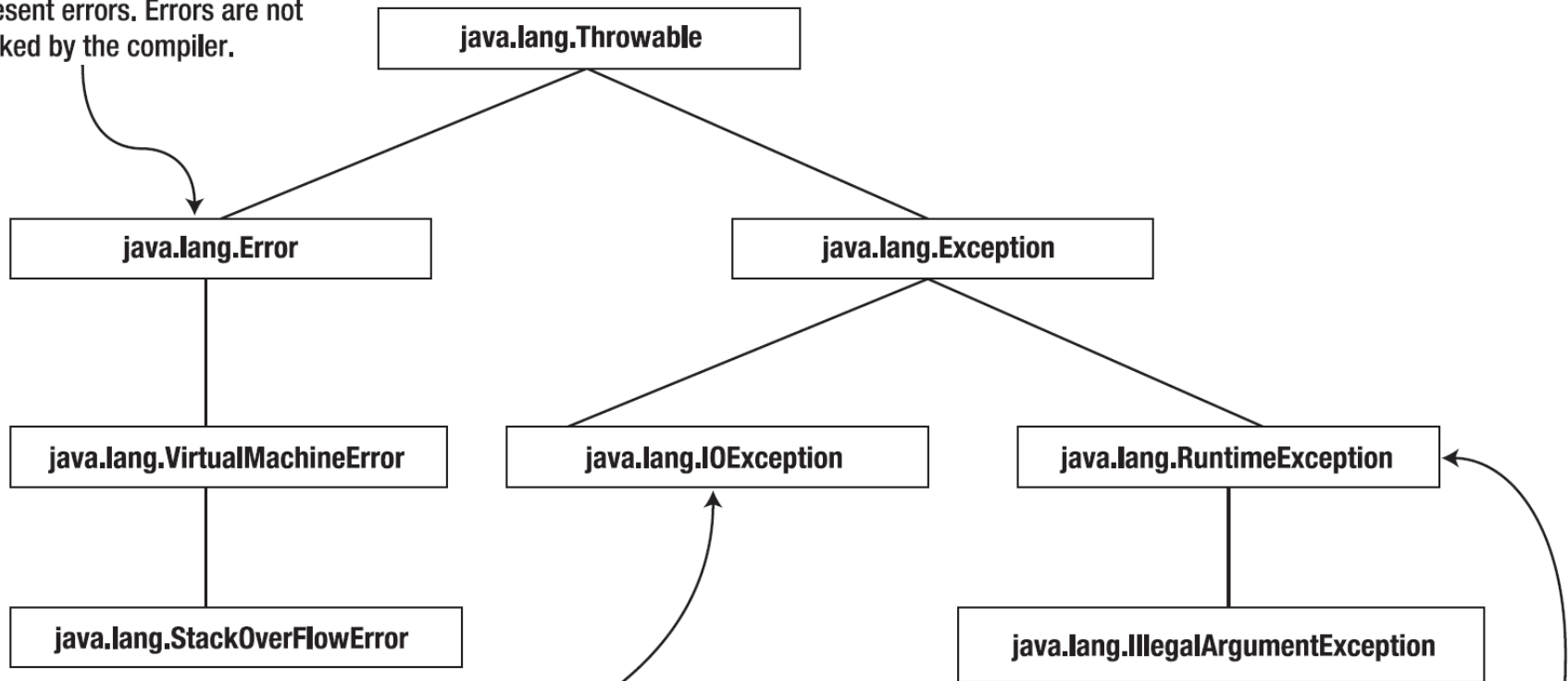
- ▶ Develop code that makes use of assertions
 - ▶ Develop code that makes use of exceptions and exception handling clauses
 - ▶ Recognize the effect of an exception arising at a specified point in a code fragment
 - ▶ Recognize situations that will result in any of the following being thrown:
ArrayIndexOutOfBoundsException,
ClassCastException, ...
- 

Understanding Exceptions in Java

- ▶ The Exception Tree in Java
- ▶ Checked Exceptions and Runtime Exceptions
- ▶ Standard Exceptions

The Exception Tree in Java

This class and its subclasses represent errors. Errors are not checked by the compiler.



An example of checked exceptions.
All subclasses of Exception and their subclasses (except RuntimeExceptions and its subclasses) are Checked exceptions. The compiler forces you to handle them or declare them.

RuntimeException and its subclasses are called runtime exceptions. The compiler does not force you to handle them or declare them.

Checked Exceptions and Runtime Exceptions

- ▶ *Handling the exception:* catch the exception and deal with it in the code
- ▶ **Checked Exceptions:**
 - instances of the Exception class, excluding the RuntimeException subtree
 - required to handle
- ▶ **Runtime Exceptions:**
 - occur due to program bugs
 - not required to provide code for these exceptions

Basics of Exception Handling

- ▶ Using the try and catch Blocks
- ▶ Using the finally Block
- ▶ Using Multiple catch Blocks

Using the try and catch Blocks

Listing 7-1. *ExceptionHandle1.java*

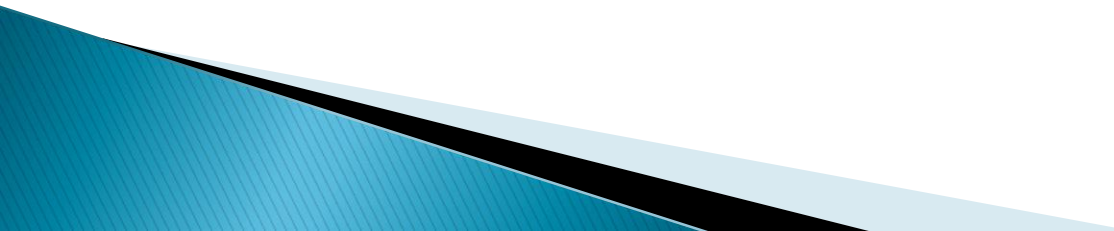
```
1. public class ExceptionHandle1{
2.     public static void main(String[] args) {
3.         int x = 15;
4.         int y = 0;
5.         try{
6.             System.out.println ("x/y: " + x/y);
7.             System.out.println("x*y: " + x*y);
8.         }
9.         catch (ArithmeticException ae) {
10.            System.out.println("An exception occurred: " + ae);
11.        }
12.        System.out.println("x-y: " + (x-y));
13.    }
14. }
```

Using the finally Block

Listing 7-2. *ExceptionHandle2.java*

```
1. public class ExceptionHandle2{
2.     public static void main(String[] args) {
3.         int x = 15;
4.         int y = 0;
5.         try{
6.             System.out.println ("x/y: " + x/y);
7.             System.out.println("x*y: " + x*y);
8.         } catch (ArrayIndexOutOfBoundsException oe) {
9.             System.out.println("An exception occurred: " + oe);
10.        }
11.        finally {
12.            System.out.println("finally block must be executed!");
13.        }
14.        System.out.println("x-y: " + (x-y));
15.    }
16. }
```


Using Multiple catch Blocks

- ▶ Only one relevant catch block, encountered first by the execution control, will be executed for a given exception.
 - ▶ The more specific catch block must precede the less specific catch block. Violation of this rule will generate a compiler error.
- 

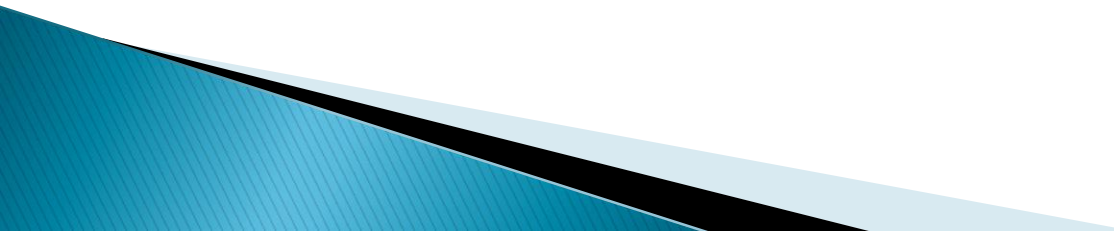
Listing 7-3. *MultipleExceptions.java*

```
1. public class MultipleExceptions{
2.     public static void main(String[] args) {
3.         int[] x = {0, 1, 2, 3, 4};
4.         try{
5.             System.out.println ("x[6]: " + x[6]);
6.             System.out.println("x[3]: " + x[3]);
7.         }
8.         catch (ArithmeticException ae) {
9.             System.out.println("An arithmetic exception occurred: " + ae);
10.        }
11.        catch (ArrayIndexOutOfBoundsException oe) {
12.            System.out.println("Array index out of bound! ");
13.        }
14.        catch (IndexOutOfBoundsException ie) {
15.            System.out.println("Some kind of index out of bound! ");
16.        }
17.        finally {
18.            System.out.println("finally block must be executed!");
19.        }
20.            System.out.println("x[0]: " + x[0]);
21.        }
22.    }
```

Rules

- ▶ If an exception is not caught by any catch block:
 - If there is no finally block, the execution stops right at the point of exception and returns to the calling method
 - If there is a finally block, the execution jumps from the point of exception to the finally block, and returns to the calling method after the finally block.

Rules (cont.)

- ▶ If the exception is caught by a catch block:
 - If there is a finally block, the execution jumps after executing the catch block to the finally block, and continues until the end of the method.
 - If there is no finally block, the execution jumps after executing the catch block to the first statement immediately after the last catch block and continues to the end of the method.
- 

Throw- ing Excep- tions

Listing 7-4. *ThrowExample.java*

```
1. public class ThrowExample{
2.     public static void main(String[] args) {
3.         double x = 15.0;
4.         double y = 0.0;
5.         try{
6.             ThrowExample te = new ThrowExample();
7.             double z = te.doubleDivide(x, y);
8.             System.out.println ("x/y: " + x/y);
9.         }
10.        catch (ArithmeticException ae) {
11.            System.out.println("An exception occurred: " + ae);
12.        }
13.            System.out.println("x-y: " + (x-y));
14.    }
15.    double doubleDivide(double x, double y) {
16.        if(y==0.0) {
17.            throw new ArithmeticException("Integer or not,
18.                please do not divide by zero!");
19.        } else {
20.            return x/y;
21.        }
22.    }
```

Declaring Exceptions

- ▶ Checked Exception: Duck It or Catch It
 - Duck it: declare it with the throws clause in the method declaration and have no try-catch blocks in the method body
 - Catch it: have try-catch blocks in the method body for that exception
- ▶ Declaring Exceptions when Overriding:
 - the exception thrown by the overriding method must be the same as, or a subclass of, the exception thrown by the overridden method

Checked Exception: Duck it

- ▶ Every checked exception that a method may throw must be declared with the throws keyword

```
public void test() throws <testException1>,  
    <testException2> {  
    // code goes here  
}
```

```
1. void callingMethod() {  
2.     calledMethod();  
3. }  
4. void calledMethod() throws IOException {  
5.     throw new IOException();  
6. }
```

Assertions

- ▶ refers to the verification of a condition that is expected to be true during the execution of a program
- ▶ can be enabled or disabled during compilation and runtime
- ▶ Syntax:

```
assert <condition>;
```

or

```
assert <condition>:<expression>;
```


Listing 7-6. *OverrideExample.java*

```
1. public class AssertionExample{
2.     public static void main(String[] args) {
3.         int x = 15;
4.         DataAccess da = new DataAccess();
5.         assert da.dataIsAccesible():"Data is not acceptable";
6.         System.out.println("Value of x: " + x);
7.     }
8. }
9. class DataAccess {
10.     public boolean dataIsAccesible(){
11.         return false;
12.     }
13. }
```

```
javac -source 1.4 AssertionExample.java
java -ea AssertionExample
```

Important Notes

- ▶ You must prepare your code for the checked exceptions: duck them or catch them.
 - ▶ If you catch an exception and recover from it the execution will resume as normal. If you throw (or re-throw) an exception from a method, you must declare it using the throws keyword in the method declaration.
 - ▶ Assertions are typically used to test and debug an application before its deployment and are not guaranteed to be enabled when the application is running.
- 