



Java Programming

3-6

Exceptions and Assertions



Objectives

This lesson covers the following topics:

- Use exception handling syntax to create reliable applications
- Use try and throw statements
- Use the catch, multi-catch, and finally statements
- Recognize common exception classes and categories
- Create custom exception and auto-closeable resources
- Test invariants by using assertions

Exceptions

- Exceptions, or run-time errors, should be handled by the programmer prior to execution.
- Handling exceptions involves one of the following:
 - Try-catch statements
 - Throw statement

Try-Catch Statements

- Try-catch statements are used to handle errors and exceptions in Java.
- In the following code example, the program will run through the try code block first.
- If no exception occurs, the program will continue through the code without executing the catch block.

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new  
        FileInputStream("missingfile.txt");  
    System.out.println("File open");  
}  
catch (Exception e) {  
    System.out.println("Something went wrong!");  
}
```

Searching for the Catch Statement

- If an exception is found, the program will search for a catch statement that catches the exception.
- If no catch statement is found, and the exception is not handled in any other way, your program will crash during run-time.

Try-Catch Statement Example

- In the code segment below, an exception can be expected if the file “missingfile.txt” does not exist (a reference is being made to a non-existent object).

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new  
        FileInputStream("missingfile.txt");  
    System.out.println("File open");  
}  
catch (Exception e) {  
    System.out.println("Something went wrong!");  
}
```

Try-Catch Statement Example (cont.)

- When the exception occurs, the catch statement is prepared to handle it by displaying the message “something went wrong”.
- The print line for “file open” will only be executed if no error is thrown.

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new  
        FileInputStream("missingfile.txt");  
    System.out.println("File open");  
}  
catch (Exception e) {  
    System.out.println("Something went wrong!");  
}
```


Action When Catch Statement is Reached

- The action that occurs when the catch statement is reached is the decision of the programmer.
- For example, the programmer could prompt the user with “File not found, please provide file name.”
- With this information, the program will be able to use another file and attempt to open that one.

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
} catch (Exception e) {  
    System.out.println("File not found, please provide file name");  
    //read file name from user input  
}
```

Using Multiple Catch Statements

- There are different kinds of exceptions and multiple catches can be used for multiple exceptions.
- This is especially useful if different exceptions need to be handled differently.
- Only one exception can be caught when multiple catch statements are used, and will be caught in the order in which they are handled.

When Multiple Catch Statements Are Effective

- Using multiple catch statements may be effective in making catch statements more specific to the certain exception that occurs.
- Multiple catch statements can be used for one try statement in order to catch more specific exceptions.

```
try {  
    //try some code that may cause an exception  
} catch (FileNotFoundException e) {  
    //code that executes when a FileNotFoundException occurs  
} catch (IOException e) {  
    //code that executes when an IOException occurs  
}
```

Using Multiple Catch Statements Example

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
    int data = in.read();  
    in.close();  
}  
catch (FileNotFoundException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
}  
catch (IOException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
}
```

FileNotFoundException may occur if "missingfile.txt" does not exist.

IOException may occur if no data is found in "missingfile.txt" when the code attempts to access it.

Using Multiple Catch Statements Example Explained

- In the previous code example, the try statement was executed first. If `FileNotFoundException` occurs, the first catch statement is triggered.
- If this exception does not occur, the program will continue to execute the try statement until it reaches the `IOException` threat.
- If the `IOException` occurs, the second catch statement is triggered.
- If no exceptions occur, the program will skip over the catch statements and continue executing the rest of the program.

Finally Clause

- Try-catch statements can optionally include a finally clause that will execute if an exception was found or not.
- Finally clauses are optional, but they will always execute no matter if an exception is caught or not.
- This is useful for including code that will execute every time the program is run - with or without an exception occurring.

Finally Clause Example

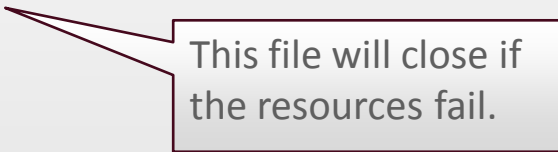
- The finally clause will execute if an exception was found or not.

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null)
            in.close();
    } catch (IOException e) {
        System.out.println("Failed to close file");
    }
}
```

Auto-Closeable Resources

- There is a “try-with-resources” statement that will automatically close resources if the resources fail.
- In this example, “missingfile.txt” will close if the try statement completes normally, or if a catch statement is executed.

```
System.out.println("About to open a file");
try (InputStream in =
    new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```



This file will close if the resources fail.

Multi-Catch Statement

- There is a multi-catch statement that allows catching of multiple exception types in the same catch clause.
- Each type should be separated with a vertical bar:

```
ShoppingCart cart = null;

try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new
     ObjectInputStream(is)) {
    cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Declaring Exceptions

- Another way to handle an exception is to declare that a method throws an exception.
- Methods can be declared to throw exceptions if they contain try statements and fail to execute correctly.
 - A try statement will go in the method declaration.
 - If the try fails, the method will throw the declared exception.

```
public static int readByteFromFile() throws IOException {  
    try (InputStream in = new FileInputStream("a.txt")) {  
        System.out.println("File open");  
        return in.read();  
    }  
}
```

Handling Declared Exceptions

- Method-declared exceptions must still be handled, but can be handled inside the method declaration or when the method is called.
- Here is an example of handling the exception when the method from the previous slide is called.

```
public static void main(String[] args) {  
    try {  
        int data = readByteFromFile();  
    } catch (IOException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Creating Custom Exceptions

- If you find that no existing exception adequately describes the exception, you can create your own, custom exceptions by extending the Exception class or one of its subclasses.

```
public class MyException extends Exception {  
    public MyException() {  
        super();  
        //MyException specific code here...  
    }  
    public MyException(String message) {  
        super(message);  
        //MyException specific code here...  
    }  
    //MyException specific code here...  
}
```

Assertions

- Assertions are a form of testing that allow you to check for correct assumptions throughout your code.
- For example, if you assume that your method will calculate a negative value, you can use an assertion.
- Assertions can be used to check internal logic of a single method:
 - Internal invariants
 - Control flow invariants
 - Class invariants

Disabling Assertions at Run-Time

- Assertions can be disabled at run-time.
- Therefore:
 - Do not use assertions to check parameters.
 - Do not use methods that can cause side effects in an assertion check.

Assertion Syntax

- There are two different assertion statements.
- If the <boolean_expression> evaluates as false, then an AssertionError is thrown.
- A second argument is optional, but can be declared and will be converted to a string to serve as a description to the AssertionError message displayed.

```
assert <boolean_expression> ;  
assert <boolean_expression> : <detail_expression> ;
```

Internal Invariants

- Internal invariants are testing values and evaluations in your methods.
- Internal invariants are to test values of variables after they've been updated or evaluated.
- They are usually used to test internal values to see if they are set or updated correctly.

```
if (x > 0) {  
    // do this  
} else {  
    assert ( x == 0 );  
    // do that  
    // what if x is negative?  
}
```


Control Flow Invariants

- Control flow invariants are assertions that can be made inside control flow statements.
- They allow you to check if code that should have been executed has not been executed.

Control Flow Invariants Example

- In the example below, since there are only four suits of cards if one of them isn't picked, there must be some other error.

```
switch (suit) {  
    case Suit.CLUBS: // ...  
        break;  
    case Suit.DIAMONDS: // ...  
        break;  
    case Suit.HEARTS: // ...  
        break;  
    case Suit.SPADES: // ...  
        break;  
    default:  
        assert false : "Unknown playing card suit";  
        break;  
}
```

Class Invariants

- A class invariant is an invariant used to evaluate the assumptions of the class instances, which is an Object in the following example:

```
public Object pop() {
    int size = this.getElementCount();
    if (size == 0) {
        throw new RuntimeException("Attempt to pop from empty
stack");
    }
    //endif

    Object result = /* code to retrieve the popped element */ ;
    // test the postcondition
    assert (this.getElementCount() == size - 1);

    return result;
}
```

Terminology

Key terms used in this lesson included:

- Assertions
- Auto-closeable statements
- Class invariant
- Control flow invariant
- Exceptions

Terminology

Key terms used in this lesson included:

- Finally clause
- Internal invariant
- Multi-catch
- Try-catch statement

Summary

In this lesson, you should have learned how to:

- Use exception handling syntax to create reliable applications
- Use try and throw statements
- Use the catch, multi-catch, and finally statements
- Recognize common exception classes and categories
- Create custom exception and auto-closeable resources
- Test invariants by using assertions

