Exceptions and Assertions

Copyright © 2012, Oracle. All rights reserved.

What Will I Learn?

Objectives

In this lesson, you will learn 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

Copyright © 2012, Oracle. All rights reserved.

Why Learn It?

Purpose

The user experience and programming functionality are very important components to a well designed program.

Imagine requesting a software from a major company, paying lots of money for it, and it breaks every time you enter the wrong input. Would that product be very successful? Probably not. The user would prefer a handler that addresses the exception and prompts the user with the issue and continues functioning.

Exceptions and assertions allow for an elegant, consistent way of debugging Java code as well handling errors that may occur throughout execution.

Copyright © 2012, Oracle. All rights reserved.

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

Copyright © 2012, Oracle. All rights reserved.

ORACLE Academy

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. In 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!");
}
```

Try-Catch Statements

If an exception is found, the program will search for a catch statement that catches the exception.

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). When the exception occurs, the catch statement is prepared to handle it by noting the user with “something went wrong”.

```
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!");
}
```

Note: If no catch statement is found, and the exception is not handled in any other way, your program will crash during run-time.

Copyright © 2012, Oracle. All rights reserved.

Try-Catch Statements

The action that occurs when the catch statement is reached is up to the programmer and how s/he decides the program should operate.

For example, rather than displaying “something went wrong” 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
}
```

Copyright © 2012, Oracle. All rights reserved.

Using Multiple Catch Statements

You may find that using multiple catch statements is very 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 possibly cause an exception
} catch (FileNotFoundException e) {
    //code that executes when a FileNotFoundException 
occurs
} catch (IOException e) {
    //code that executes when an IOException occurs
}
```

Copyright © 2012, Oracle. All rights reserved.

Using Multiple Catch Statements

In the code segment below, the try statement is executed first. There are 2 possible threats for exceptions:

1. FileNotFoundException – this may occur if “missingfile.txt” does not exist
2. IOException – this may occur if no data is found in “missingfile.txt” when the code attempts to access it.

```
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();
}
//continued on next slide
```

Threat #1

Threat #2

Using Multiple Catch Statements

If FileNotFoundException occurs, the first catch statement is triggered.

If this exception does not occur, the program will continue to execute in 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.

```
//continued from previous slide...
 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");
}
```

Copyright © 2012, Oracle. All rights reserved.

Finally Clause

Try-Catch statements can optionally include a finally clause that will always 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");
    }
}
```

Copyright © 2012, Oracle. All rights reserved.

Auto-closeable Resources

There is a “try-with-resources” statement that will automatically close resources if the resources fail. In the following 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());
}
```

Copyright © 2012, Oracle. All rights reserved.

Multi-Catch Statement

There's a multi-catch statement that allows you to catch 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);
}
```

Copyright © 2012, Oracle. All rights reserved.

Declaring Exceptions

Another way to handle an exception is to declare that a method *throws* an exception.

- 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();
    }
}
```

Copyright © 2012, Oracle. All rights reserved.

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());
    }
}
```

Copyright © 2012, Oracle. All rights reserved.

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.

The code may look something like this:

```
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...
}
```

Copyright © 2012, Oracle. All rights reserved.

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
- 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.

Copyright © 2012, Oracle. All rights reserved.

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.

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

Copyright © 2012, Oracle. All rights reserved.

Control Flow Invariants

Assertions can be made inside control flow statements such as shown below. These are called control flow invariants.

```
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;
}
```

Copyright © 2012, Oracle. All rights reserved.

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");
}

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

    return result;
}
```

Copyright © 2012, Oracle. All rights reserved.

Terminology

Key terms used in this lesson included:

Exceptions
Try-catch statement
Multi-catch
Finally clause
Auto-closeable statements
Assertions
Internal Invariant
Control Flow Invariant
Class Invariant

Copyright © 2012, Oracle. All rights reserved.

Summary

In this lesson, you 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

Copyright © 2012, Oracle. All rights reserved.

Practice

The exercises for this lesson cover the following topics:

- Using try-catch statements
- Creating and using exceptions
- Using assertions

Copyright © 2012, Oracle. All rights reserved.