2020

Object-Oriented Programming

Topic Notes: Exception Handling (Java)

- *Errors* are what causes a program to produce unexpected results.
- An *error* may either cause a program to produce unexpected results or terminate the execution of a program or even cause the system to crash!

Types of *Errors*

1. **Compile-time errors**
2. **Run-time errors**

Compile-time Errors

- These kind of errors are detected and displayed by the Java compiler.
- If a *compile-time error* occurs in a program, the program will not compile, i.e the `javac` (the Java compiler) will not create a `.class` file for your `.java` source code.

The most common types of compile-time errors are

- Missing semicolons
- Missing or mismatched brackets in classes and methods
- Misspelling of identifiers or keywords
- Missing double quotes in Strings
- Using an undeclared variable
- Incompatible types in assignments
- Bad reference to objects
- use of `=` in place of `==` operator

Example

```
public class CompileTimeErrorDemo {
    public static void main(String[] args) {
        System.out.println("Hello, World!")  // We have purposely missed a
semicolon here
    }
}
// ------------------------------------------------------------------------
--------
/*  The compiler will generate the following error message:
*
*   CompileTimeErrorDemo.java: 3 : ';' expected
*   System.out.println("Hello, World!")
*   ^
```

```
*   1 Error
*
*/
```

Run-time Error

- A program may compile successfully (create the `.class` file) but may not produce expected results.
- It may be due to improper implementation of business logic or errors such as stack overflow.

The most common types of run-time errors are

- Dividing an integer by zero
- Accessing an element that is out of bounds of an array
- Trying to store a value into an array of an incompatible type
- Passing a parameter that is not in a valid range or type for a method
- Attempting to use negative size for an array
- Converting an invalid string to a number
- Accessing a character that is out of bounds of a String

```
public class RumTimeErrorDemo {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int c = 5;
        int x = a / (b - c);  // division by zero!
        System.out.println("x = " + x);
        int y = a / (b + c);
        System.out.println("y = " + y);
    }
}
```

Exceptions

- An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing by zero, it creates an exception object and throws it.
- Four steps for error handling
    - find the problem (**Hit** the exception)
    - inform that an error occurred (**throw** the exception)
    - receive the error information (**catch** the exception)
    - correct the problem (**handle** the exception)
- Common Java Exceptions

Exception Type	Cause
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes

Exception Type	Cause
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between Strings and numbers fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when a program tries to perform an action not allowed by the security setting
StackOverFlowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a String

Syntax for Exception Handling in Java

```
try {
    // statements that might cause an exception
} catch (ExceptionType e) {
    // statements to process the exception caused
}
```

Example

```
class ExceptionHandlingDemo {
    public static void main(String[] args) {
        int a = 10;
    }
}
```

Note

- You can place multiple nested `try` statements

```
class NestedTryDemo {
    public static void main(String[] args) {
        try {
            int a = 2, b = 4, c = 2, x = 7, z;
            int[] p = {2};
            p[3] = 22;
```

```
            try {
                z = x / ((b * b) - (4 * a * c));
                System.out.println("The value of z = " + z);
            } catch (ArithmeticException aE) {
                System.out.println("Division by zero in Arithmetic
 Expression!");
            }
        } catch (ArrayIndexOutOfBoundsException aI) {
            System.out.println("Array index is out of bounds!");
        }
    }
}
```

- You can also place multiple catch statements

```
class MultipleCatchDemo {
    public static void main(String[] args) {
            int[] a = {5, 10};
            int b = 5;
        try {
            int x = a[2] / b - a[1];
        } catch (ArithmeticException aE) {
            System.out.println("Division by zero!");
        } catch (ArrayIndexOutOfBoundsException aI) {
            System.out.println("Array Index Out Of Bound!");
        } catch (ArrayStoreException aS) {
            System.out.println("Wrong Data Type In Array!");
        }
        int y = a[1] / a[0];
        System.out.println("y = " + y);
    }
}
```

The `finally` statement

- The finally statement can catch any exception that was not caught by any of the previous `catch` statements, much like the `default` label in the `switch-case` construct.
- The `finally` statement will *always* execute, even in no exception was thrown in the code.
- We generally use the `finally` statement to perform clean-up operations after the code, ex - closing files and releasing system resources.
- The general syntax of a `finally` block is as follows

```
class A {
    public static void main(String[] args){
        try {
            int a = 4;
            int b = 5 / (a - 4);
        } catch (ArithmeticException aE) {
```

```
            System.out.println("Arithmetic Exception");
        } finally{
            System.out.println("Program Continues!");
        }
    }
}
```

Throwing our own custom exceptions

```
import java.lang.Exception;
class MyCustomException extends Exception{
    MyCustomException(String message) {
        super(message);
    }
}
class ExceptionTest {
    public static void main(String[] args){
        int x = 5;
        int y = 1000;
        try {
            float z = (float) x / (float) y;
            if (z < 0.01) {
                throw new MyCustomException("Number is too small");
            }
        } catch (MyCustomException mce) {
            System.out.print("my.custom.Exception: ");
            System.out.println(mce.getMessage());
        } finally {
            System.out.println("This is always printed!");
        }
    }
}
```