
ORACLE
Academy

Java Programming

2-4

Exceptions and Assertions

ORACLE
Academy

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 (errors that happen during the execution of the code), should be handled by the programmer prior to program execution
- Handling exceptions involves one of the following:
  - Try-catch statements
  - Throw statement

It is easy to catch common small errors such as mathematical mistakes, failure to open files and failure to calculate information correctly.
The key thing to remember is that errors are not expected behaviour and should not be expected behaviour.

Try-Catch Statements

- Try-catch statements are used to handle errors and exceptions in Java and follow these steps:
  - 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
  - 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

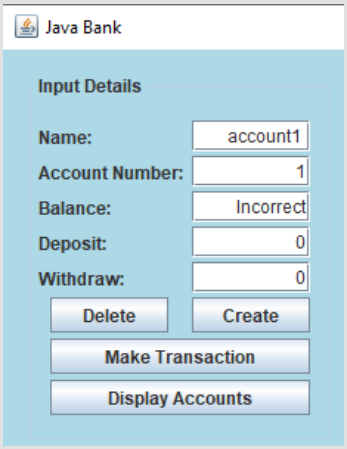
Error handling in the JavaBank

Currently there is no run-time error handling in the JavaBank application!

a) Run the `JavaBank.java` file

b) Enter an account name and number and then enter “Incorrect” as the value for the balance

c) Click the Create button and then evaluate the exception thread in the console



Error handling in the JavaBank

d) The following error message was returned:

```
Exception in thread "AWT-EventQueue-0"
java.lang.NumberFormatException: For input string:"Incorrect"
```

e) It explains that the value “Incorrect” caused a NumberFormatException error that the program could not deal with at run-time

f) This is because the code currently accepts the text input from the TextField and tries to parse it to an Integer value

```
balance = Integer.parseInt(balanceJTextField.getText());
```

g) If a value that cannot be parsed to a whole number is entered a run-time error occurs

Adding Error handling

- Add the following to the code that creates a new account

1. Write a try statement that encloses the text field input except for the account name (already a String)

```
try {
   if (accountNumJTextField.getText() == "0") {
      accountNum = 0; }
   else {
      accountNum = Integer.parseInt(accountNumJTextField.getText());
   }//endif
   if (balanceJTextField.getText() == "0") {
      balance = 0;    }
   else {
      balance = Integer.parseInt(balanceJTextField.getText());
   }//endif
}//endtry
```

Adding Error handling

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

2. Add a catch statement after the closing try bracket to catch any exception and display it to the console

```
}//endtry
catch(Exception e) {
    System.out.println(e);
}//end catch
```

Displays the type of exception and the input that caused it.

- The catch statement is where the error is handled

It is important to understand that code in the try statement will be executed until an error occurs. **If no error occurs the catch statement is not executed!**

Searching for the Catch Statement

- If no catch statement is found, and the exception is not handled in any other way, your program will crash during run-time
  - You can choose how you want to deal with the error by supplying an appropriate error message or providing a default value for a variable
  - whatever suits the error that has been dealt with

Although your program will not crash with a try catch statement it is important to remember that an error has occurred, and your program has not executed successfully.

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
- Using multiple catch statements may be effective in making catch statements more specific to the certain exception that occurs

When Multiple Catch Statements Are Effective

- Multiple catch statements can be used for one try statement in order to catch more specific exceptions

3. A NumberFormatException has already been identified by the current catch statement, add this new catch statement above the existing one:

```
}//endtry
catch(NumberFormatException e) {
   name=("");
   JOptionPane.showMessageDialog(null, "Incorrect numeric value entered." );
}//end catch
catch(Exception e) {
```

The **Exception** e catch statement must come last as it will catch all errors!

Using Multiple Catch Statements Example

```
try {
   if (accountNumJTextField.getText() == "0") {
      accountNum = 0; }
   else {
      accountNum = Integer.parseInt(accountNumJTextField.getText());}//endif
   if (balanceJTextField.getText() == "0") {
      balance = 0;  }
   else {
       balance = Integer.parseInt(balanceJTextField.getText());}//endif
}//endtry
catch(NumberFormatException e) {
   name=("");
   JOptionPane.showMessageDialog(null, "Incorrect numeric value entered." );
}//end catch
catch(Exception e) {
   System.out.println(e);
}//end catch
```

NumberFormatException will occur if an incorrect type is entered.

Exception will occur if any exception other than a NumberFormatException error occurs.

Only one exception is caught when multiple catch statements are used, and will be caught in the order they are handled in.

Using Multiple Catch Statements Example Explained

- In the previous code example, the try statement was executed first. If a NumberFormatException 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 end, or an unknown exception occurs
- If an unknown exception 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

Be careful! A finally clause in a try catch does not operate like a default in a switch statement. The finally clause will always be executed regardless of whether an error is caused.

Finally Clause Example

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

4. Add a finally clause after the final catch statement, move the code from the end of the method that clears the text fields into the finally block

```
}//end catch
finally {
   // clear the JTextFields for new data
   nameJTextField.setText(" ");
   accountNumJTextField.setText("0");
   balanceJTextField.setText("0");
   depositJTextField.setText("0");
   withdrawJTextField.setText("0");
}//end finally
```

The finally block ensures that the text fields are reset to their default values regardless of a successful creation of an account.

The finally clause is always executed. You must ensure this is what you want to happen when you create your program!

Auto-Closeable Resources

- There is a “try-with-resources” statement that will automatically close resources if the resources fail
  - The try-with-resources statement is a try statement that declares one or more resources
  - A resource is an object that must be closed after the program is finished with it
  - The try-with-resources statement ensures that each resource is closed at the end of the statement

Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

Auto-Closeable Resources

- 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());
}//end try catch
```

This file will close if the resources fail.

Don't worry if you don't understand the file manipulation code, it will be covered later in the course!

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:

```
catch(NumberFormatException | InputMismatchException e) {
   name=("");
   JOptionPane.showMessageDialog(null, "Incorrect numeric value entered." );
}//end catch
catch(Exception e) {
   System.out.println(e);
}//end catch
```

5. Add this to your try catch statement so that you can catch both a NumberFormatException and an InputMismatchException in the same catch statement

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();
    }//endtry
}//end method readByteFromFile
```

Handling Declared Exceptions

- Method-declared exceptions must still be handled, but can be handled inside the method declaration or from where the method was 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();
    }//endtry
    catch (IOException e) {
       System.out.println(e.getMessage());
    }//endcatch
}//end method main
```

The exception is handled in the calling method (main) not the method where the exception occurred (readByteFromFile).

Creating Custom Exceptions

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

```
public class MyException extends Exception {

   public MyException(String message) {
      super(message);
   }//end constructor method

   //MyException specific code here...

}//end class MyException
```

Calling Custom Exceptions

- A custom exception is called from within an existing Exception statements catch block
- To work with your own custom exception you must create a new object of that exception class as part of a throw statement
- A throw statement passes the error back to the calling statement
- A throws statement must be added to the end of the method signature identifying the type of Exception that will be returned

Calling Custom Exceptions

- The myException example on a previous slide requires a message to be passed into its constructor, you need to pass the error string into the object when you create it

```
private void actionMethod() throws MyException {
   //method code
   try {
      //method code
   }
   catch(Exception e) {
      throw new MyException("An unhandled error occurred!!");
   }//end catch
```

Calling Custom Exceptions

- The calling function must also be wrapped in a try catch statement so that the custom Exception can be processed

```
try {
   actionMethod();
}catch (MyException e) {
   System.out.println(e);
}//end try catch
```

- The error message can then be displayed

```
try {
   actionMethod();
}catch (MyException e) {
   JOptionPane.showMessageDialog(null, e );
}//end try catch
```

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
- An assert statement is used to declare an expected boolean condition in a program
- If the program is running with assertions enabled, then the condition is checked at runtime

Assertions allow users to be able to test the functionality of their code. Assertions can be very useful in debugging your code.

Assertions

- Assertions can be used to check internal logic of a single method:
  - Internal invariants
    - Internal Invariants allow you to check that your code deals with values correctly
  - Control flow invariants
    - Control flow invariants usually allow you to check if something in your code that should have been executed hasn't been
  - Class invariants
    - A class invariant can specify the relationships among multiple attributes and should be true before and after any method completes

Disabling Assertions at Run-Time

- Assertions can easily be disabled at run time so that you can disable all of your test features when you run the code
- Therefore:
  - Do not use assertions to check parameters
  - Do not use methods that can cause side effects in an assertion check

Assertions are not usually used in production code for reasons mentioned above. They can also slow down the program at runtime. Most IDEs have assertions disabled by default.

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?
}//endif
```

Internal Invariants Example

1. Create the following program that converts a double to an int, what could go wrong?

```
package assertions;
import java.util.Scanner;
public class AssertEx {
//program that will convert a double to an int
    public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter a number: ");
    double x = in.nextDouble();
    in.close();

    System.out.println("Value of x: " + x);
    int y = (int) (x);

    System.out.println(y);
    }//end method main
}//end class AssertEx
```

Internal Invariants Example

2. Execute the code from the previous slide and test with the following values:
   - 300000
   - 3000000000

What happens to the values when they are converted and then displayed to screen?

Internal Invariants Example

- 300000

```
Please enter a number: 300000
Value of x: 300000.0
300000
```

- 3000000000

```
Please enter a number: 3000000000
Value of x: 3.0E9
2147483647
```

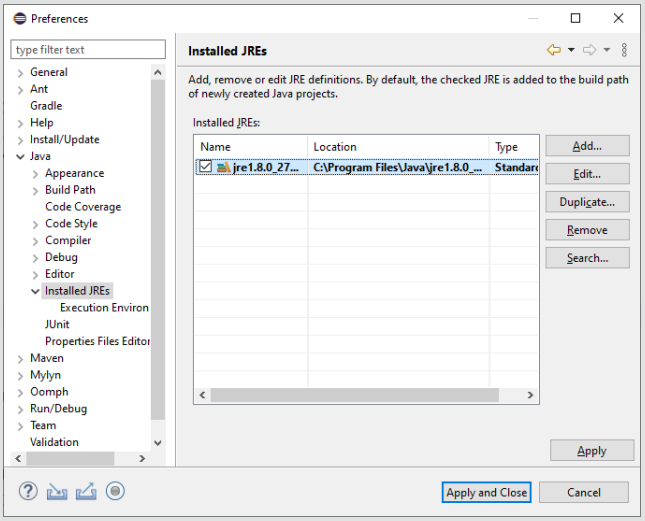
Data will be lost during the during lossy conversion between double and int if the number is too large

Internal Invariants Example

- You could always write additional code that would warn you if the number was too large for the conversion
- You could use a series of output statements to display all the values to the console
- These solutions would have to be removed before packaging up your application
- Any forgotten test code would lead to erroneous output for the user once the software was released
- Easier to use assertions!

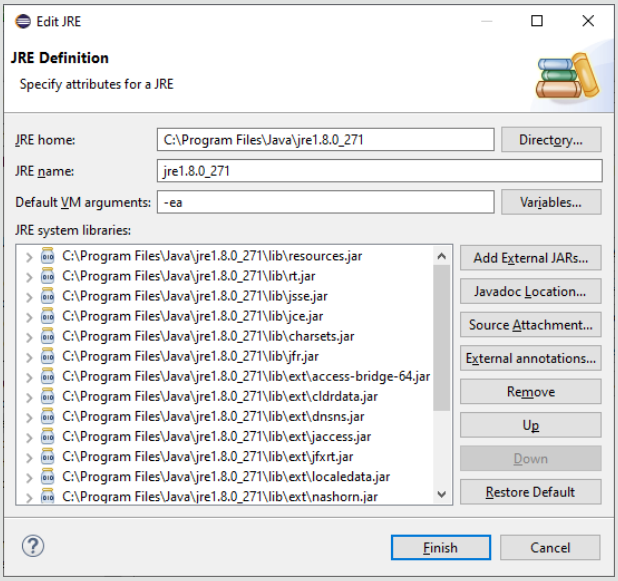
Internal Invariants Example

3. Assertions are turned off by default in Eclipse
4. To turn them on follow these steps:
   - Window
   - Preferences
   - Java
   - Installed JREs
   - Select the current JRE
   - Click the Edit button



Internal Invariants Example

5. In the Edit window do the following:
   - Click into the Default VM arguments box
   - Add –ea as the value (-ea stand for enable assertions)
   - Click Finish
   - In the preferences window click Apply and Close



Internal Invariants Example

6. Update your code to include an assert statement

```
in.close();

assert x < 2_147_483_647.0 : "Number too large to convert to integer";
/*assert evaluates the Boolean expression
  If the expression evaluates to true the normal code continues,
  If the expression evaluates to false it will throw an
  AssertionsError exception*/

System.out.println("Value of x: " + x);
```

7. Run the code using 3000000000 as the input value

assert x < 2_147_483_647.0 is the Boolean expression
: "Number too large to convert to integer" is the optional error message that will be returned if the Boolean expression returns false

Internal Invariants Example

8. The assert statement created an Exception, and the error was displayed

```
Please enter a number: 3000000000000000
Exception in thread "main" java.lang.AssertionError: Number too large to convert to integer
        at assertions.AssertEx.main(AssertEx.java:13)
```

9. Redo steps 4 and 5 of this exercise to turn assertions back off within Eclipse by removing the –ea command
10. Rerun your code the assert statement is now ignored

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
- They also allow you to evaluate what happens if values you do not think are possible were to be processed by your code
- There is an example on the next slide…

Control Flow Invariants Example

- In this example there should only be four suits of cards, if any other value is provided there must be an error

```
public static void main(String[] args) {
    String suit = "Card";
    switch (suit) {
        case "Clubs"     : System.out.println("Clubs");
                           break;
        case "Diamonds"  : System.out.println("Diamonds");
                           break;
        case "Hearts"    : System.out.println("Hearts");
                           break;
        case "Spades"    : System.out.println("Spades");
                           break;
        default          : assert false : "Unknown playing card suit";
                           break;
    }//end switch
}//end method main
```

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;
}//end Object pop
```

A class invariant should be true before and after any method completes. Therefore it can be used to ensure that the object is behaving in the way expected.

Control Flow Invariants

- The assert statement will return a Boolean value once it has been executed
- Internal Invariants allow you to check that your code deals with values correctly.
- By asserting values internally in your code you can test that your code works with a range of values
- A class invariant can specify the relationships among multiple attributes and should be true before and after any method completes
- Therefore it can be used to ensure that the object is behaving in the way expected when it is being used within the program

Terminology

- Key terms used in this lesson included:
  - Assertions
  - Auto-closeable statements
  - Class/Internal invariant
  - Control flow invariant
  - Exceptions
    - Try-catch statement
    - Multi-catch
    - Finally clause

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


ORACLE
Academy