



---

Lab 5 – Exception Handling

---

## Objectives:

- Learn the concept of exception handling.
- Use Java exception handling to write more robust code.

## Exception

An **exception** is an event, which occurs during the execution of a program, which disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "something" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see Figure 1).

The call stack showing three method calls, where the first method called has the exception handler.

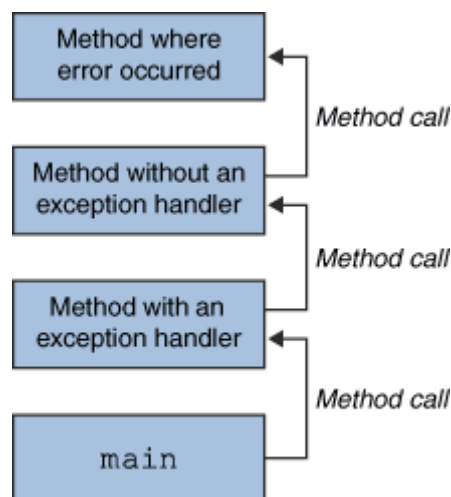


Figure 1 The call stack

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in

the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler (see Figure 2.)

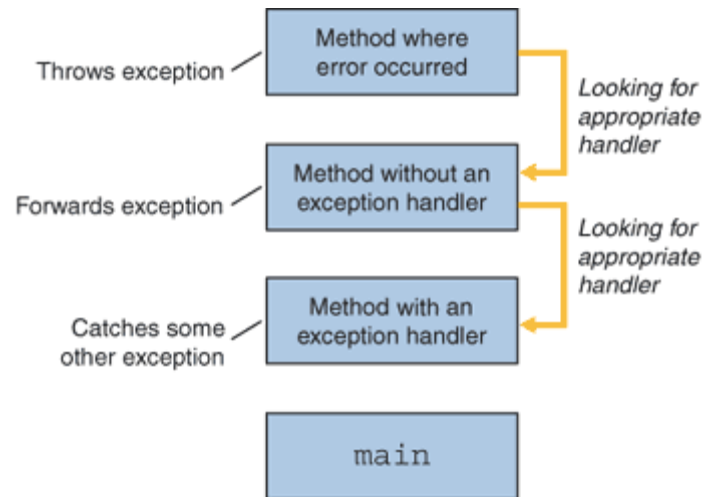


Figure 2 Searching the call stack for exception handler

One of the most important advantages of exception is that it separates the error-handling code from regular code.

In traditional programming, error detection could result in spaghetti code (complex and my branching constructs.) For example, consider the pseudocode that reads an entire file into the memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

There might be some possible errors, what happen if:

- the file cannot be opened?
- the file length cannot be determined?
- enough memory cannot be allocated?
- the read fails?
- the file cannot be closed?

To handle these types of errors, some error detection codes must be added to the program as show below:

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
```

```

determine the length of the file;
if (gotTheFileLength) {
    allocate that much memory;
    if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) {
            errorCode = -1;
        }
    } else {
        errorCode = -2;
    }
} else {
    errorCode = -3;
}
close the file;
if (theFileDintClose && errorCode == 0) {
    errorCode = -4;
} else {
    errorCode = errorCode and -4;
}
} else {
    errorCode = -5;
}
return errorCode;
}

```

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile` function used exceptions instead of traditional error-management techniques, it would look more like the following.

```

readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed){
        doSomething;
    }
}

```

## Java Exception Hierarchy

Java exceptions are all classes those inherit from `Throwable` as shown in Figure 3.

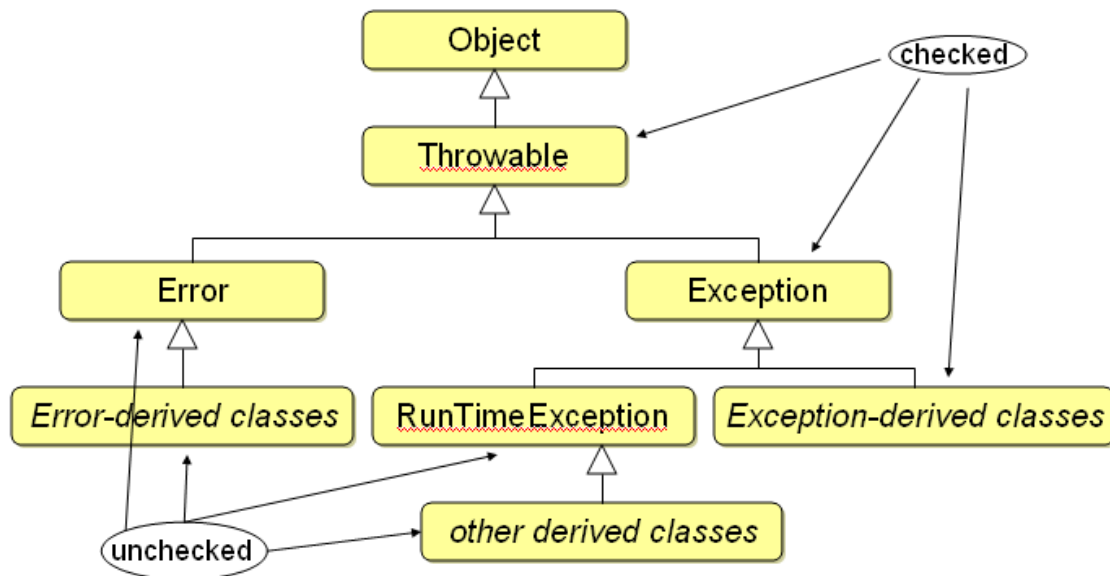


Figure 3 Java Exception Hierarchy

There two type of exception, **checked** and **unchecked** exception.

The **checked exceptions** (all classed derived from `Exception`) must be handled. That is, a `try-catch` must either be nested around the call to the method that throws the exception or the method signature must include a `throws` clause for this exception. (Then other methods that invoke this method must then catch the exception or throw it up the line.) The compiler will throw an error message if it detects an uncaught exception and will not compile the file.

The **unchecked exceptions**, all `Error`'s and `RunTimeException`, do not have to be caught because these type of exceptions indicate programming bugs. This avoids requiring that a `try-catch` be place around, for example, every integer division operation to catch a divide by zero or around every array variable to watch for indices going out of bounds. So the programmers must ensure that these types of exception must be prevented.

However, you should handle possible run-time exceptions if you think there is a reasonable chance of one occurring.

### The Catch and Specify Requirement

Whenever you call a method that declares to throw any types of checked exception(s), the program must follow that catch and specify requirement. Otherwise, a compilation error will occur indicating that there is an unhandled exception. This means that code that throw certain exceptions must be enclosed by either of the following:

- A `try` statement that catches the exception. The try must provide a handler for the exception, as described in *Catching and Handling Exceptions*.
- A method that specifies that it can throw the exception. The method must provide a `throws` clause that lists the exception, as described in *Specifying the Exceptions Thrown by a Method*.

## Catch and Handling Exceptions

This section describes how to use the three exception handler components — the `try`, `catch`, and `finally` blocks — to write an exception handler.

The following example defines and implements a class named `FileReadingDemo` that reads a text file specified in the command-line argument and prints the reading text to the console.



```
import java.io.*;
public class FileReadingDemo {

    public static void main(String[] args) {
        String filename = args[0];
        FileReader file = new FileReader(args[0]);
        BufferedReader buffer = new BufferedReader(file);
        String txt = "";
        while ((txt = buffer.readLine()) != null) {
            System.out.println(txt);
        }
        buffer.close();
    }
}
```



### Your turn

Complete the following steps.

- 
1. Create new Java project called `lab5`.
  2. Setup the project to run JUnit as described in **Lab 4**.
  3. Create a new class named `FileReadingDemo` and copy the code above. You will notice that there are three errors as show in Figure 4 because there are unhandled checked exceptions. The IDE shows errors by marking  at the lines that cause compilation errors.

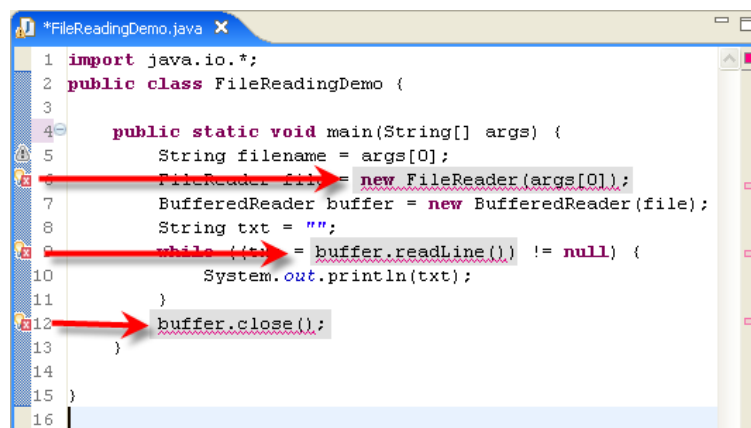


Figure 4 Errors when not handling checked exceptions



4. Open Java API document to see the description of class `FileReader` in package `java.io`. Look at the constructor of `FileReader` which takes a string as its argument. In the constructor signature, it is defined as follow:

```
public FileReader(String fileName) throws FileNotFoundException
```

That is for the first error. The second and third errors belong to method `readLine` and `close` in class `BufferedReader`. Take a look in Java API document for the method `readLine` and `close` in class `BufferedReader`. The signatures are defined as follow:

```
public String readLine() throws IOException
```

```
public void close() throws IOException
```

In order to handle these exceptions, we must understand the exception handling mechanism.

### The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a `try` block. In general, a `try` block looks like the following.

```
try {  
    code ← could thrown a checked exception  
}  
    catch and finally blocks . . .
```

The segment in the example labeled code contains one or more legal lines of code that could throw an exception. (The `catch` and `finally` blocks are explained later.)

To construct an exception handler for the constructor method from the `FileReader` class, enclose the exception-throwing statements of the constructor method within a `try` block. There is more than one way to do this. You can put each line of code that might throw an exception within its own `try` block and provide separate exception handlers for each. Or, you can put all the code within a single `try` block and associate multiple handlers with it. The following listing uses one `try` block for the entire method because the code in question is very short.



5. Add the `try` block enclose the code that will throw exceptions

```
try {  
    FileReader file = new FileReader(args[0]);  
    BufferedReader buffer = new BufferedReader(file);  
    String txt = "";  
    while ((txt = buffer.readLine()) != null) {  
        System.out.println(txt);  
    }  
    buffer.close();  
}  
// catch and finally statements ...
```

If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. To associate an exception handler with a `try` block, you must put a `catch` block after it.


## The catch Blocks

You associate exception handlers with a `try` block by providing one or more `catch` blocks directly after the `try` block. No code can be between the end of the `try` block and the beginning of the first `catch` block.

```
try {  
    } catch (ExceptionType1 name) {  
    } catch (ExceptionType2 name) {  
    }
```


Each `catch` block is an exception handler and handles the type of exception indicated by its argument. The argument type, `ExceptionType`, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class. The handler can refer to the exception with `name`.

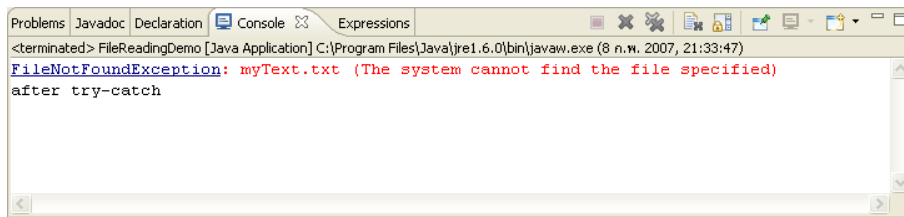
The `catch` block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

 6. Add the following catch blocks to handle the methods that throw exception.

```
try {  
    .  
    .  
    .  
} catch (FileNotFoundException e) {  
    System.out.println("FileNotFoundException: " + e.getMessage());  
} catch (IOException e) {  
    System.out.println("Caught IOException: " + e.getMessage());  
}  
System.out.println("after try-catch");
```

Both handlers print an error message. If the exception thrown is `FileNotFoundException`, it matches the first `catch` block. After all statements in the block have been executed, the execution skips the remaining `catch` block(s) and continues to the next statement after `try-catch` block.

 7. Run the program with the argument as a non-existing file. See the result compare to Figure 5.




```
<terminated> FileReadingDemo [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (8 n.w. 2007, 21:33:47)
FileNotFoundException: myText.txt (The system cannot find the file specified)
after try-catch
```

Figure 5 FileNotFoundException output

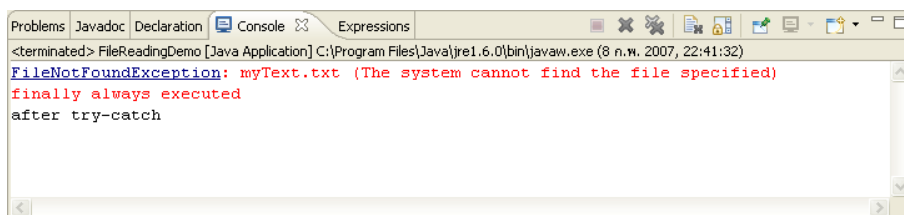
## The finally Block

The `finally` block always executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. But `finally` is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`. Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated.

- 
8. Add the following `finally` block after the last `catch` block, and run.


```
. . .
} catch (IOException e) {
    System.out.println("Caught IOException: " + e.getMessage());
} finally {
    System.out.println("finally always executed");
}
```

The result should be similar to Figure 6.



```
<terminated> FileReadingDemo [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (8 n.w. 2007, 22:41:32)
FileNotFoundException: myText.txt (The system cannot find the file specified)
finally always executed
after try-catch
```

Figure 6 Finally output with exception

- 
9. Change the command-line argument to an existing text file. Run and see the result compare to Figure 7.

The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. So it's the perfect place to perform cleanup.



```

Problems Javadoc Declaration Console Expressions
<terminated> FileReadingDemo [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (8 n.w. 2007, 22:47:19)
import java.io.*;

public class FileReadingDemo {

    public static void main(String[] args) {
        String filename = args[0];
        try {
            FileReader file = new FileReader(args[0]);
            BufferedReader buffer = new BufferedReader(file);
            String txt = "";
            while ((txt = buffer.readLine()) != null) {
                System.out.println(txt);
            }
        } catch (FileNotFoundException e) {
            System.err.println("FileNotFoundException: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            System.err.println("finally always executed");
        }
        System.err.println("after try-catch");
    }
}

finally always executed
after try-catch

```

Figure 7 Finally output without exception

## Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the main method in the `FileReadingDemo` class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception.

If the main method doesn't catch the checked exceptions that can occur within it, the main method must specify that it can throw these exceptions. Let's modify the original main method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the main method that won't compile.

```

public static void main(String[] args) {
    String filename = args[0];
    FileReader file = new FileReader(args[0]);
    BufferedReader buffer = new BufferedReader(file);
    String txt = "";
    while ((txt = buffer.readLine()) != null) {
        System.out.println(txt);
    }
    buffer.close();
}

```

To specify that main can throw two exceptions, add a throws clause to the method declaration for the main method. **(This is what we have done so far when we write Java statement that need to access I/O such as file or keyboard input. We do not want the handle the exception, so we add throws IOException clause after main.)** The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```

public static void main(String[] args) throws IOException,
    FileNotFoundException {

```



10. Create new class name `ThrowExceptionDemo` and copy the following code:

```
public static void main(String[] args) throws IOException,
    FileNotFoundException {
    String filename = args[0];
    FileReader file = new FileReader(args[0]);
    BufferedReader buffer = new BufferedReader(file);
    String txt = "";
    while ((txt = buffer.readLine()) != null) {
        System.out.println(txt);
    }
    buffer.close();
}
```



11. Run it with existing file and non-existing file. Compare the output with Figure 8 & Figure 9.

Figure 8 FileNotFoundException

Figure 9 Without exception

### How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception. Regardless of what throws the exception, it's always thrown with the `throw` statement.

The Java platform provides numerous exception classes. All the classes are descendants of the `Throwable` class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write.

## The throw Statement

All methods use the `throw` statement to throw an exception. The `throw` statement requires a single argument: a throwable object. `Throwable` objects are instances of any subclass of the `Throwable` class. Here's an example of a `throw` statement.

```
throw someThrowableObject;
```

## Throwable Class and Its Subclasses

The objects that inherit from the `Throwable` class include direct descendants (objects that inherit directly from the `Throwable` class) and indirect descendants (objects that inherit from children or grandchildren of the `Throwable` class). Figure 10 illustrates the class hierarchy of the `Throwable` class and its most significant subclasses. As you can see, `Throwable` has two direct descendants: `Error` and `Exception`.

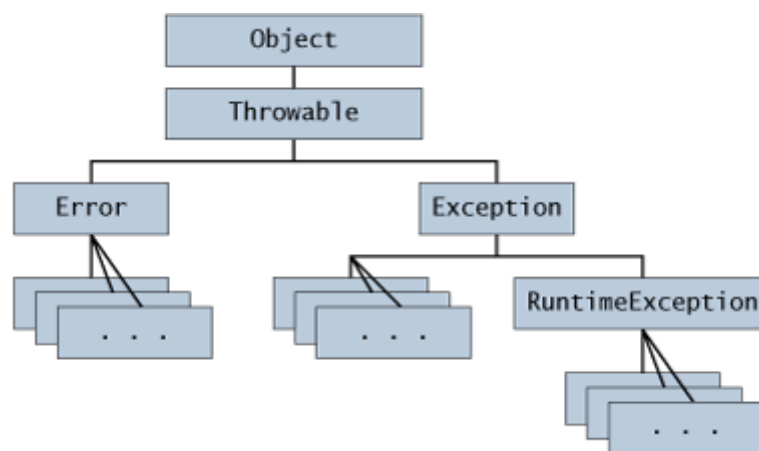


Figure 10 The `Throwable` class

## Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an `Error`. Simple programs typically do not `catch` or `throw` `Errors`.

## Exception Class

Most programs `throw` and `catch` objects that derive from the `Exception` class. An `Exception` indicates that a problem occurred, but it is not a serious system problem. Most programs you write will `throw` and `catch` `Exceptions` as opposed to `Errors`.

The Java platform defines the many descendants of the `Exception` class. These descendants indicate various types of exceptions that can occur. For example, `IllegalAccessError` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

One `Exception` subclass, `RuntimeException`, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is `NullPointerException`, which occurs when a method tries to access a member of an object through a null reference.

## Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?

To create new exception, define a new class which extends from `Exception`, and have `Exception` as its postfix. For example, to create a new exception called `RangeException`:

```
public class RangeException extends Exception {
    public RangeException(String msg) {
        super(msg);
    }
    public RangeException() {
    }
}
```



### ***Your turn (1)***

Complete the class `ArrayOfInt` as specified.

Suppose you are writing a class called `ArrayOfInt` which has two attributes, `element` (array of `int`) and `size` (`int`). This class implements a growable array of `int` values. The capacity of the array can be grown (increased) or shrunk (decreased) (new capacity can be greater or smaller than the current capacity).

- `element` stores integers (when first create, it stores nothing).
- `size` stores how many integers `element` currently stores.

This class must have the following members:

- Constructors:
  - with one argument, the initial capacity. This constructor initializes `element` to an array of `int` with its length equals the initial capacity.
  - with no argument that initializes `element` to an array of `int` with its length equals 10, the default initial capacity.
- Methods:
  - `setCapacity()`: takes one `int` input argument, the new capacity. This method must do the following tasks:
    - First, check that the new capacity is different from the current capacity. If the new capacity equals to current capacity, this method does nothing.
    - Second, allocate a new array of `int` with its length equals the new capacity.

- Third, copy the content of `element` to the new array.
  - Finally, assign this new array to `element`.
- `get()`: takes one `int` input argument, the index of the element to be retrieved by the caller. This method must check that the index is in the valid range ( $0 \leq \text{index} < \text{size}$ ). If the index is valid, this method returns an `int` stored at the specified index. Otherwise, it throws `InvalidElementException`.
- `add()`: takes one `int` input argument, the value to be stored (added to the array). This value will be appended to the end of `element`. The variable `size` should also be updated to an appropriate value. Before appending the new value to the end of `element`, it must check whether `size` exceeds the capacity (the length of `element`) or not. If the capacity is exceeded, this method must create a new `element`, which is a new array with twice its previous capacity. In the end, `element` will store the original content appended with the new value.
- `set()`: takes two `int` input arguments, index and value. This method must check that the index is in the valid range ( $0 \leq \text{index} < \text{size}$ ). If the index is valid, this method set the element at the specified index to that input value. Otherwise, it throws `InvalidElementException`.

1. Import `lab5.jar` into your project.
2. There are errors indicated that `InvalidElementException` has not been declared. Create a new exception by creating a new class that extends from `java.lang.Exception` as the code below:

```
public class InvalidElementException extends Exception {
    public InvalidElementException(String msg) {
        super(msg);
    }

    public InvalidElementException() {
    }
}
```

The only thing required in the class is its constructors.

3. Create new Java test case for `ArrayOfInt` as in Lab 4, and select methods `setCapacity`, `get`, `add`, and `set` as the methods to be tested.
4. In `ArrayOfIntTest.java` add two data members, and modify the `setUp` method as follow:

```
public class ArrayOfIntTest {
    ArrayOfInt emptyList;
    ArrayOfInt aList;
    @Before
    protected void setUp() throws Exception {
        int[] a = { 1, 2, 3 };
        emptyList = new ArrayOfInt();
        aList = new ArrayOfInt(a);
    }
}
```

Method `setUp` will be run before each test. We will use `setUp` to initialize our test data that will be available for each test. In each test in Lab 4, we had to create a lot of same data over and over. We will start with two `ArrayOfInt`, `aList` which has 3 elements, and `emptyList`.

Method `tearDown` will be the clean up method that run after each test. In our situation, we don't have anything to clean up so we don't have to modify `tearDown`.

5. Start adding the first test, modify `testSetCapacity` method with the following code:

```
@Test
public void testSetCapacity() {
    assertEquals(emptyList.element.length,
                 ArrayOfInt.DEFAULT_INITIAL_CAPACITY);
    aList.setCapacity(5);
    assertEquals(5, aList.element.length);
    aList.setCapacity(20);
    assertEquals(20, aList.element.length);
}
```

We test that the `emptyList` must have the element's length equals to `DEFAULT_INITIAL_CAPACITY`, 10, since the capacity of our `ArrayOfInt` is the `element.length` attribute. Set the capacity to some more different values, and compare with expected values.

6. Modify the `testGet` method, and add new test `testGetWithException` method as follow.

```
@Test
public void testGet() throws InvalidElementException {
    int data;
    data = aList.get(0);
    assertEquals(1, data);
    data = aList.get(1);
    assertEquals(2, data);
    data = aList.get(2);
    assertEquals(3, data);
}

@Test(expected = InvalidElementException.class)
public void testGetWithException() throws InvalidElementException {
    int data = aList.get(3);
}
```

We know that element 0, 1, 2 of `aList` has value 1, 2, and 3 respectively. The statements

```
data = aList.get(3);
```

must throw `InvalidElementException` since the index exceed the capacity. The line

```
@Test(expected=InvalidElementException.class)
```

is added before the method header to test that the method has to throw `InvalidElementException`, otherwise the test will fail.

7. Modify `testAdd` method as follow:

```
@Test
public void testAdd() {
    emptyList.add(4);
    assertEquals(4, emptyList.get(0));
    assertEquals(1, emptyList.size());
    emptyList.add(10);
    assertEquals(10, emptyList.get(1));
    assertEquals(2, emptyList.size());
    aList.add(99);
    assertEquals(99, aList.get(3));
    assertEquals(4, aList.size());
}
```

8. Modify `testSet` method, and add `testSetWithException` method as follow:

```
@Test
public void testSet() {
    int data;

    data = aList.get(0);
    aList.set(0, data + 1);
    assertEquals(aList.get(0), data + 1);
    assertFalse(aList.get(0) == data);
}

@Test(expected=InvalidElementException.class)
public void testSetWithException() {

    emptyList.set(3, 5);
}
```

9. Implement the `ArrayOfInt` class for method, `setCapacity`, `get`, `add`, and `set` until all tests are passed.



### ***Your turn (2)***

1. Create new package called `util`.
2. Create new class called `Keyboard` in package `util`.
3. In `Keyboard.java`, add four static methods
  - a. `readInt()`
  - b. `readInt(String prompt)`
  - c. `readDouble()`
  - d. `readDouble(String prompt)`
4. In each method, implement the method in the way that it will display prompt message and wait for user input and then convert that input into the appropriate data types (`readInt` converts to `int`, and `readDouble` convert to `double`). If the user input cannot be converted to its desired type, the method must try to get new user input until user enters the correct number format. An appropriate error message should be displayed.  
Note: Look at Java API document for class `Integer`, and `Double`, handle some exceptions when the input is not in the correct format and I/O error occurs.

5. Export to util.jar and save in your desktop.
6. Create new java project.
7. Go to project properties.
8. Select Java Build Path > Libraries > Add External JARs...
9. Select util.jar from step (5) in your desktop then click OK.
10. Create a Java application.
11. Add statement import util.Keyboard.
12. In main method, get some user input using Keyboard.readXXX.
13. Run the application to test it.
14. Now you have utility for reading keyboard input. Keep the file for future use.

## Reference

*Sun's Java Tutorial*, <http://java.sun.com/docs/books/tutorial/essential/exceptions/>.





---

### Lab 5 - Exception Handling.

---

<b>Task</b>	<b>Description</b>	<b>Result</b>	<b>Note</b>
1	Create all test cases		
2	<code>setCapacity</code>		
3	<code>get</code>		
4	<code>add</code>		
5	<code>set</code>		
6	Create <code>util.jar</code> <ul style="list-style-type: none"><li>• <code>class Keyboard</code></li></ul>		
7	Use <code>util.Keyboard</code>		